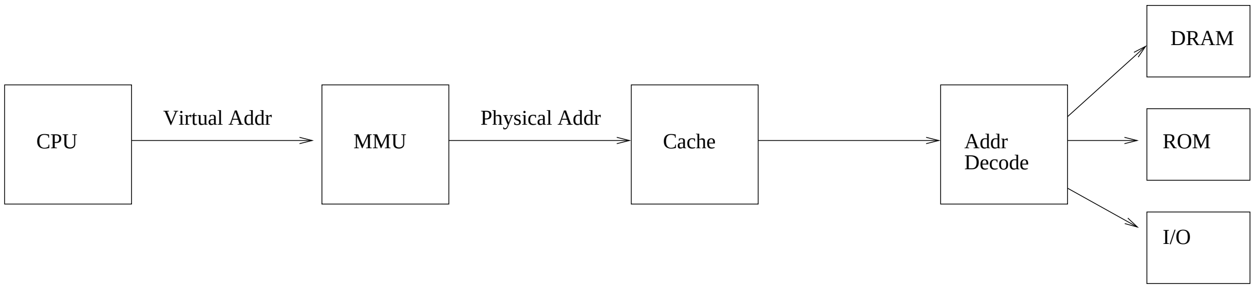
Virtual Memory

We have stated that a UNIX process is a virtual computer in which a thread of execution (virtual processor) runs within a private virtual address space. In this unit, we will begin to explore the mechanisms by which this is implemented, both from a generic hardware and a generic kernel standpoint, and with specific X86 examples.

All addresses used in a program are **virtual addresses**. Before being used to address physical RAM, ROM, or memory-mapped I/O, virtual addresses are **translated** to **physical addresses**. This translation is performed **by hardware** within the processor known as the **Memory Management Unit (MMU)**.



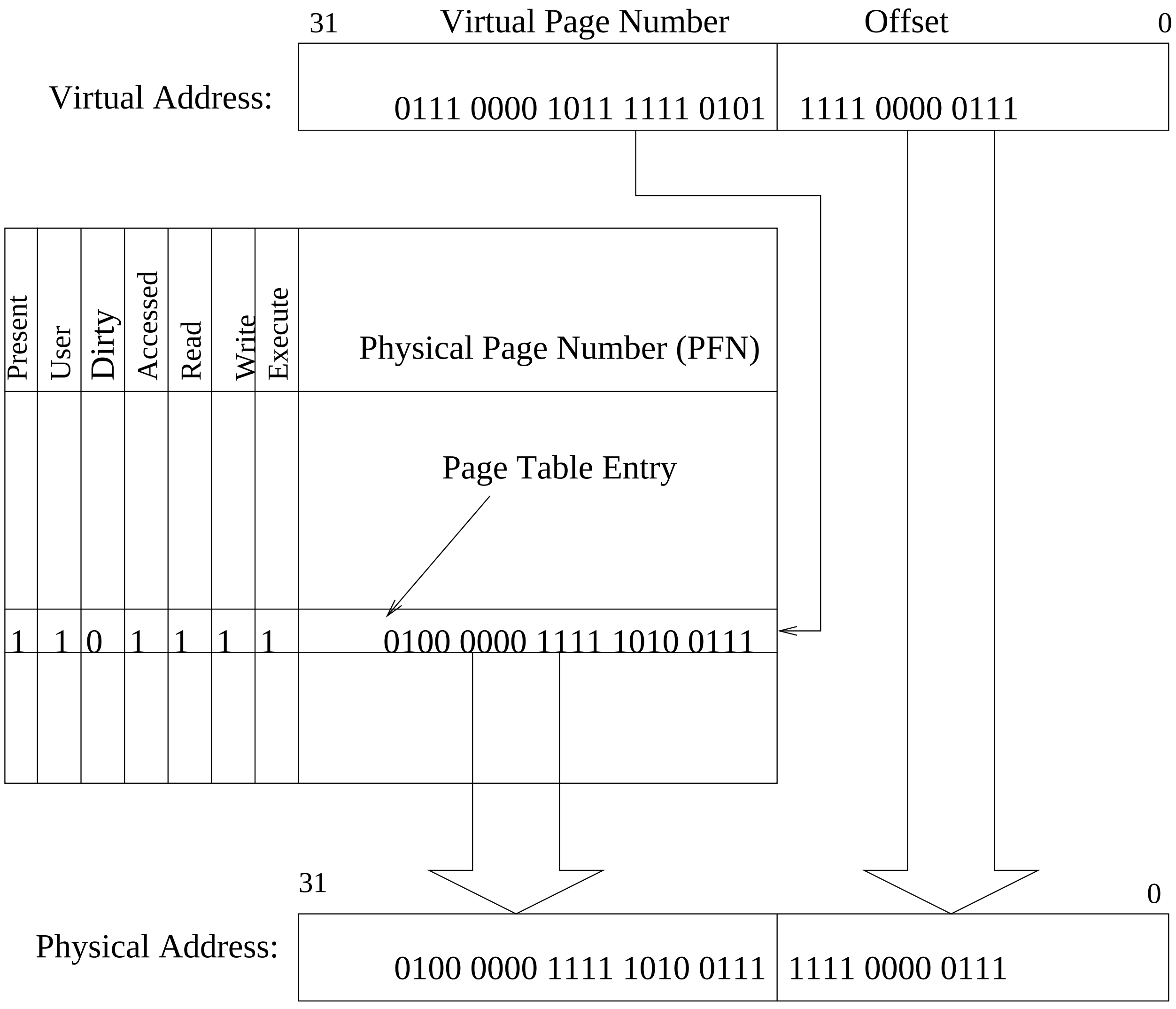
Since the operating system kernel has sole access to the MMU’s lookup tables, it has the ability to completely control the view of memory a given process can see.

The Page Table Entry (PTE)

Address translation is performed with the granularity of a **page**. An address is divided into two parts: The **page number**, comprising the most significant bits, and the **offset**, comprising the least significant bits. Address translation operates on the **virtual page number**, replacing it with the **physical page number**. The offset is passed through unchanged. Typical page sizes are 2K, 4K and 8K. On the X86 32-bit and 64-bit architectures a page size of 4K is used.

Another name for a physical page of memory is a **page frame**, because we can think of physical pages as empty frames into which the actual meaningful content is placed from time to time. We’ll see that a given page frame will hold many different virtual page images over time.

As a first cut conceptual model, we can think of the Page Table as a monolithic array of Page Table Entries (PTEs). The starting physical address of this array is given by a CPU register, called generically the "BASE" register (on X86 architecture, this will be register CR3). Then the page number portion of the virtual address is the index into this array that selects a specific PTE. Each PTE controls the translation of one page of virtual memory. The PTE contains the page number portion of the translated physical address (the Page Frame Number of PFN), and the several bitwise flags, which are illustrated *conceptually* below:



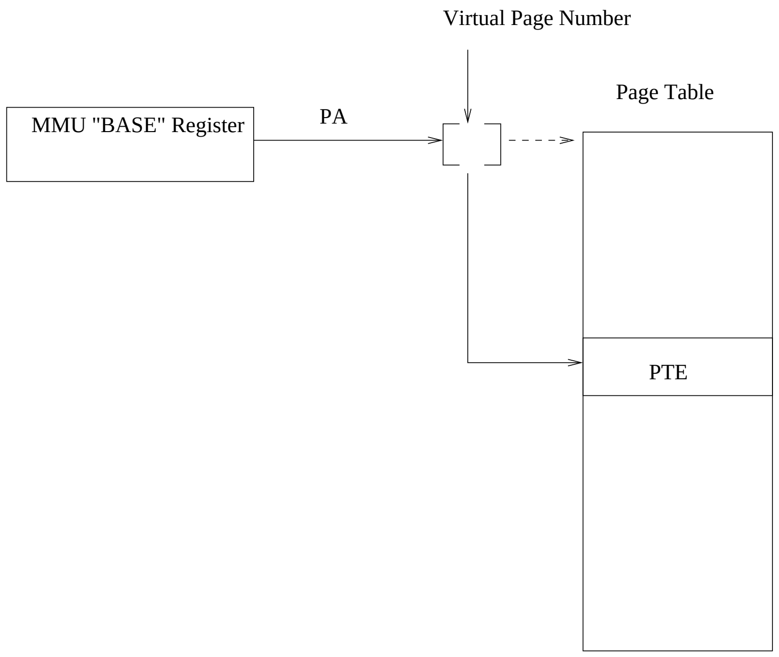
- **Present bit**: If set, this virtual page is resident (maps to a physical page frame given by the PFN field) and the rest of the bits of the PTE are also valid. If the Present bit is clear, there is no physical page frame allocated to this virtual address. An attempt to access this page results in a a **Page Fault**. As we will see, this is not necessarily a bad thing.

- **Protection bits**: Determines what type of accesses are allowed to this page: read, write, execute. An attempt to perform a disallowed access results in a **protection fault**. (Not all hardware maintains all three of these protection bits. In particular, the x86 architecture prior to Pentium-4 has only two levels of access control: readonly and readwrite, and does not distinguish between protection faults and page faults.)

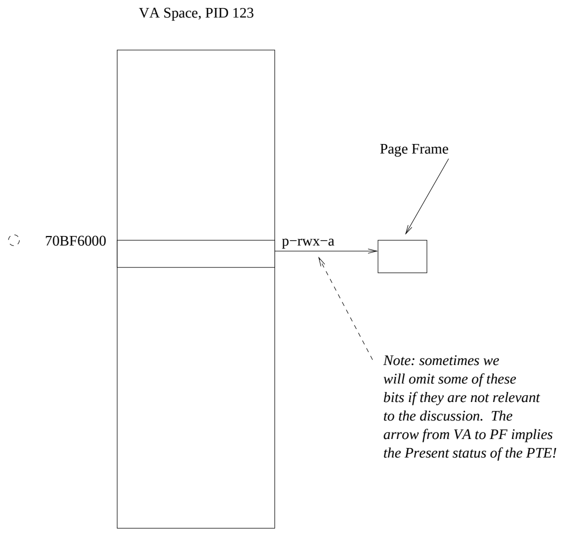
- **Dirty bit**: Set by hardware when a page is written to **via this PTE**. Cleared by the kernel when the page has been synced to backing storage.

- **Accessed bit**: Set by hardware when a page is accessed (read, write or execute) **via this PTE**. Cleared by the kernel while aging (scanning) page frames for re-use.

- **User/Supervisor Bit**: When set, this page can be accessed while the processor is in user mode. If clear, the processor must be in Supervisor (aka "kernel") mode to access the page. This allows kernel memory to be mapped into the virtual address space of a process and be visible only when the process enters the kernel. The Linux X86 architecture uses the User/Supervisor Bit approach, but other architectures switch to a different set of page tables upon entry to the kernel. (*Note 1: recent security vulnerabilities in X86 processors such as Spectre and Meltdown have led to a different strategy where pages tables are swapped on entry to the kernel. Note 2: Many architectures provide for more than just two levels of access control, e.g. X86 provides 4 levels or "rings".*)



Another way we can diagram address translation is to represent a virtual address space of a process as a box, with a series of arrows showing the mapping of given virtual pages to page frames. The presence of the arrow implies the existence of a PTE with a P bit set and a valid PFN. Above the arrow we can write additional flag bits as needed. Below, we see a translation of virtual page 70BF6 (spanning virtual addresses 70BF6000-70BF6FFF) to a page frame (the PFN is not stated), with the Present, Read, Write, Execute and Access bits of the PTE set to 1.


VA Space, PID 123
Page Frame
70BF6000
p−rwx−a
Note: sometimes we will omit some of these bits if they are not relevant to the discussion. The arrow from VA to PF implies the Present status of the PTE!

Multi-Level Page Tables

Conceptually, as depicted above, the virtual page number is an index into an array of Page Table Entries (PTEs), located at a specific physical address. There are several problems which prevent this from being a realistic implementation.

Let us consider a 32-bit architecture with 4K pages, therefore the page numbers are 20 bits long and there are 2ˆ20 possible page numbers. If each PTE is 32 bits long, then each page table would consume 4MB. Although this might seem like a small amount of memory, remember that each process on the system has its own page table. A system with 100 processes would therefore be wasting a large amount of memory on page tables.

The problem is even more severe on 64-bit machines which have a larger virtual address space. Even if only 48 bits of the address space are recognized, with a 4K page, there would be 2ˆ36 PTEs (each PTE must be 64 bits to handle the longer addresses) consuming 2ˆ39, or 512GB, of physical memory per process!

The problem of large page tables is addressed with **multi-level** page tables. The physical address of the top-level page table is given in the CPU/MMU "BASE" register (register %cr3 on X86). The most significant bits of the virtual address index into this top-level table. The result is not the page table entry, but is the physical address of the corresponding table at the next lower level of the hierarchy. That physical address, plus the bits from the next most significant part of the virtual address, locates the entry in the next page table, etc. At the last step, the actual PTE (page table entry) is fetched, which contains the PFN and the flags.

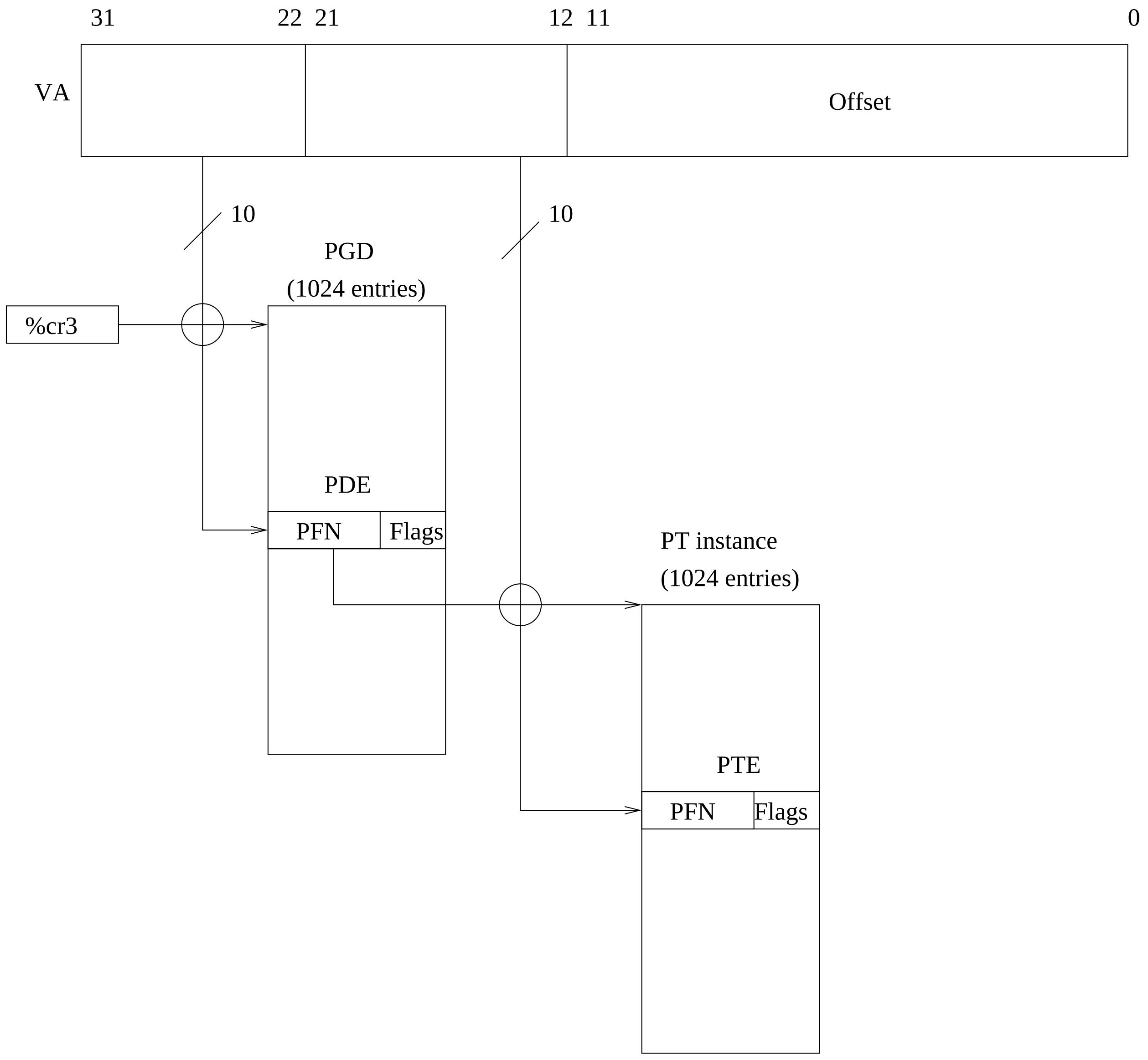
On modern computer architectures, 2-, 3- or 4-level paging structures are the norm. 2-level or 3-level structures are typical for 32 bit machines, while 64 bit virtual addresses mandate 4-level structures to achieve any kind of memory efficiency.

Page tables are in effect a data structure, kept in physical memory. The layout of this data structure is mandated by the processor architecture. The kernel has been compiled with architecture-specific code that understands the layout of the page tables and can manipulate and manage them.

X86-32, 2-level page structure

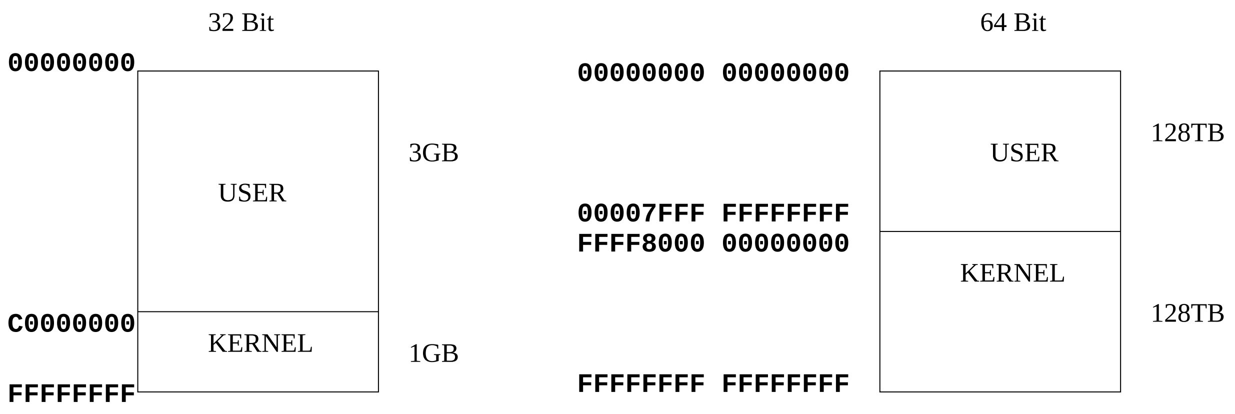
On the X86 32 bit system, the page table is 2-level with the page number split 10/10 bits. The %cr3 special control register in the CPU (which must be running in Supervisor mode to change it) gives the 32-bit physical memory address of the start of the top-level table. The Linux kernel calls this the Page Global Directory (PGD). The PGD is an array of 1024 entries, each 32 bits long, and each entry (known as a PDE: Page Directory Entry) is in the same format as a Page Table Entry (PTE). However, the Page Frame Number (PFN) field of the PDE is a pointer to the next level of the 2-level structure. Each PGD entry represents 2ˆ22 (4MB) of virtual address space.

We see that the PGD consumes 4096 bytes, or one physical page. The next level, the Page Table (PT), contains 1024 PTE (Page Table Entries) each 32 bits (4 bytes) long, thus each PT also fits within a 4K page. The virtual address space and the physical address space are both 32 bits, or 4GB. *There is a mode called Physical Address Extension (PAE), which we will not cover here, that makes physical addresses 64 bits so more than 4GB of physical RAM can be accessed. However, in PAE mode, virtual addresses are still limited to 32 bits. PAE is rarely used since most modern computers would operate in 64 bit mode.*



Address space layout, 32 and 64 bit Linux

The Linux kernel reserves a portion of the virtual address space for itself. We’ll see in subsequent units that while each user-mode process has a private virtual address space, the kernel works inside a shared kernel virtual address space. Therefore, the complete virtual address space is not available for user-mode use. In the 32-bit environment, the kernel reserves 1GB, leaving 3GB for user mode. When we go to 64 bits, the split is 50/50, with each getting 128TB. This is illustrated below:



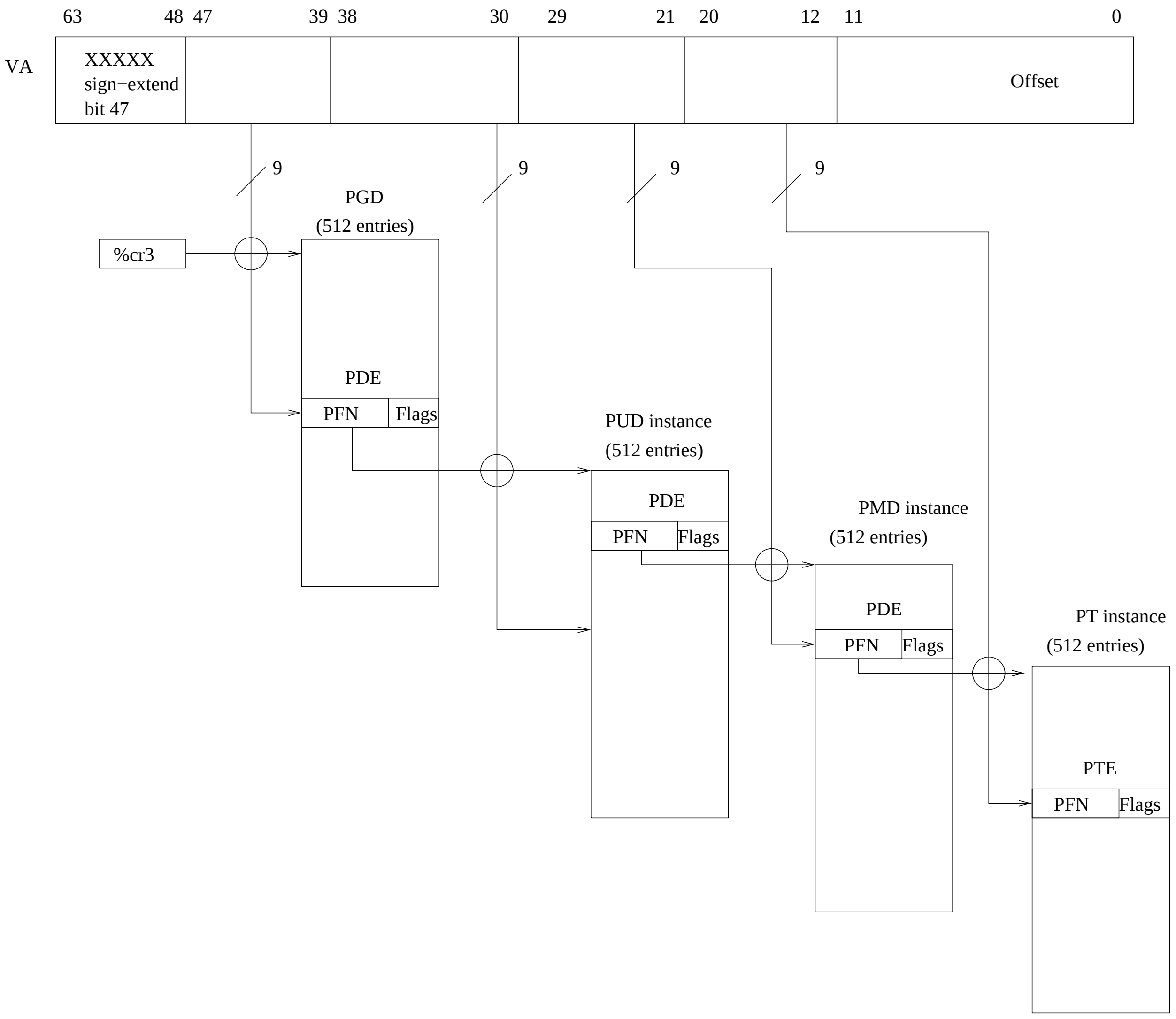
X86-64, 4-level page structure

Under the X86 64 bit architecture, registers are 64 bits wide. Virtual addresses however are actually only 48 bits wide, and are sign-extended into the remaining 16 bits. This makes addressing somewhat confusing. E.g. valid addresses jump from 0x0000 7FFF FFFF FFFF to 0xFFFF 8000 0000 0000 (spaces inserted for clarity). Therefore, a given virtual address space is limited to "only" 256TB. This point at which the addresses become "negative" is the dividing line in the Linux x86-64 kernel between user and kernel virtual memory. As with the 32-bit model, the kernel takes the upper portion. Now however the user process has 2ˆ47=128TB of virtual address space and the kernel also has 128TB. The Intel X86-64 architecture supports up to 52 bits (a 40-bit PFN) of physical addressing which is 4PB, although contemporary implementations use a much smaller physical address bus size.

Page size continues to be 4K to retain compatibility with older code. Therefore we have 48-12 = 36 bits remaining as a virtual page number. This is split evenly into four levels (9/9/9/9). Each PTE must be 64 bits wide since 32 bits would not be enough room to encode the 40-bit PFN. Therefore within one 4K table we get 2ˆ9=512 entries.

The %cr3 register (now 64 bits wide) points to the PGD, as with the 32-bit architecture. Entries in the PGD give the PFNs pointing to instances of the next level, which is called Page Upper Directory (PUD). The PUD in turn contains PFN pointers to instances of Page Middle Directories (PMD), which finally contain the pointers to the Page Tables which contain the PTEs.

Each entry of the PGD in 64 bit mode now references 2ˆ39=512 GB of address space. Each PUD entry references 2ˆ30=1GB, each PMD entry 2ˆ21=2MB and finally each PTE 2ˆ12=4KB.



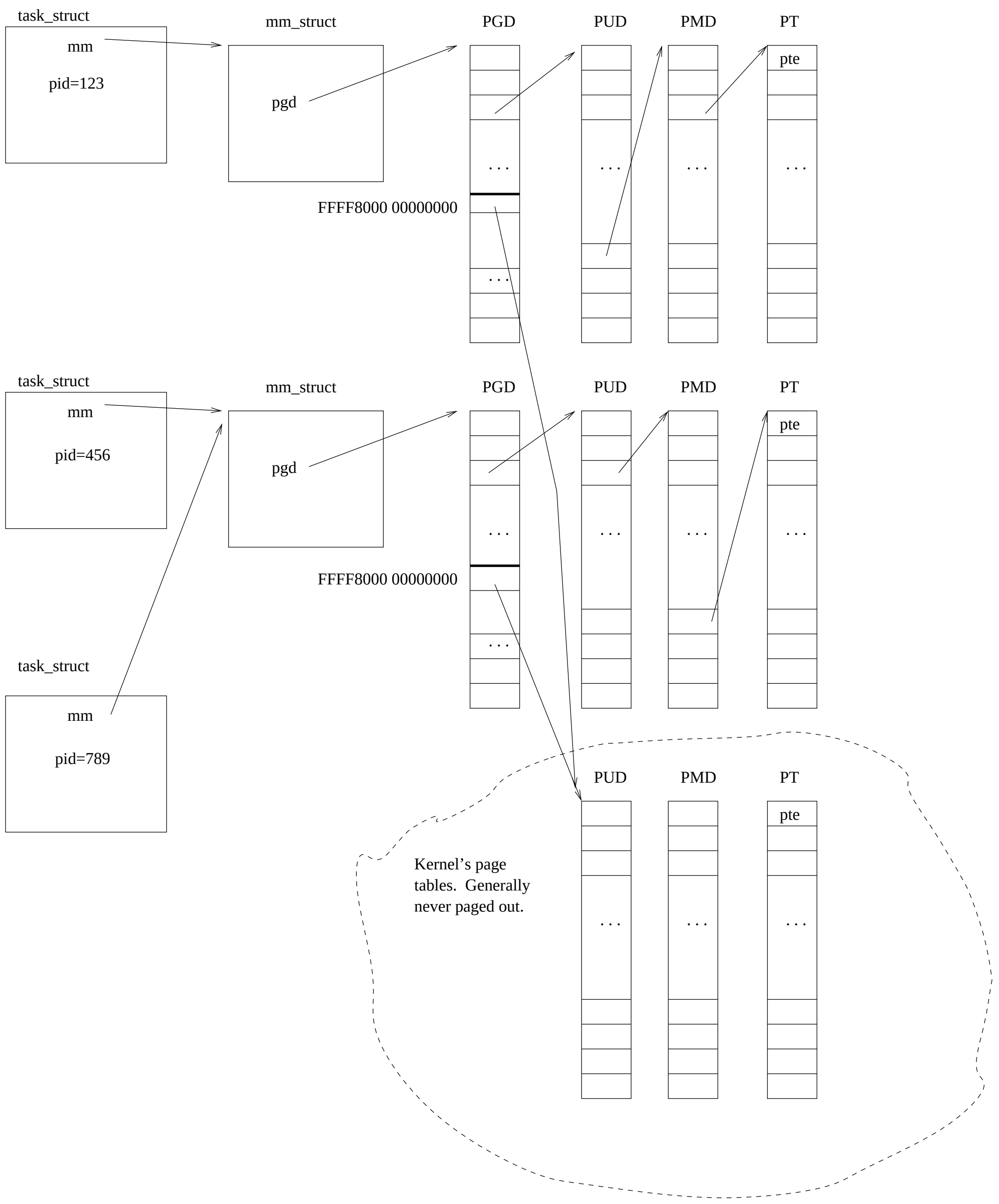
Sparse and Shared Multi-Level Page Tables

It is possible for entries in Page Directory tables (PGD/PUD/PM) to contain "NULL pointers". This means that the entire contiguous range of virtual addresses corresponding to that entry is not mapped. This "NULL" pointer is implemented by having the Present bit of the entry set to 0. Whenever such a NULL pointer is encountered during the traversal of the multi-level page table structure, it causes a Page Fault, the same as if the Present bit of the PTE were 0.

The Linux kernel uses this advantageously in performing "just in time" page table allocations. All processes get allocated an individual PGD. However, the lower-level tables hanging off the PGD entries are not allocated until the corresponding areas of virtual memory are created (or grown) AND a page fault happens (see more under Page

Fault Resolution). Therefore, areas of virtual memory which are not used (these can be quite large, especially with a 64-bit system) do not consume additional resources. This is analogous to sparse file allocation (see unit #2).

It is also possible for a given Page Table or PUD or PMD to be pointed at by more than one PGD. The Linux kernel uses this trick so it can create just one instance of the multi-level page table structure for the shared kernel memory, and allow all processes to share that (see more under Linux Address Space). In the illustration below, 456 and 789 are two tasks (threads) that are within the same process address space. We also see the sharing of kernel mode page tables. (*Note: this is changed somewhat due to recent security concerns, see appendix*)



Translations Cache & Context Switch

Page table entries reside in main memory, and thus consume two important resources: The memory needed to store the page tables and the time required to perform translations. We have seen how multi-level page tables address the former concern. The latter is extremely significant too, since a translation must be performed on every memory access. If the MMU worked literally as described thus far, for a 2-level page table, 2 additional memory accesses would be required for every access attempted, i.e. memory performance would be reduced to one third. For a 4-level table, it would be reduced to one fifth! Therefore, a cache, often called the **Address Translation Cache (ATC)** or **Translation Lookaside Buffer (TLB)**, is used to speed translations.

Like any cache, the ATC/TLB is a content-addressable memory. A **key** (or "tag") is presented, and either the stored **value** is accessed (a **"hit**"), or the key is not found in the cache, which is known as a **cache miss**. Recall that the address translation process takes the page number portion of a virtual address and returns a page table entry. Therefore, the key in this case is the virtual address being translated (or more correctly, the page number portion of the VA), and the value accessed is the PTE.

For every virtual memory access, the ATC/TLB is searched to see if a valid, cached PTE exists for the referenced virtual page number. If so, the translation is performed using the cached page table entry. Otherwise, hardware "walks" the page table structure and loads the correct translation, keeping it cached for possible future re-use. The cache is of finite (and typically rather small) size, so the insertion of a new cached translation into the ATC generally eradicates an older cache entry to make room.

A **context switch** is when the kernel changes from one process to another which implies a change in both the task and the address space. (in contrast, a **task switch**, at least in the Linux kernel, could be a change from one thread to another while staying in the same address space). To perform a context switch on X86, the kernel simply modifies the %cr3 register with the physical address of the PGD representing the new process. However, this creates a problem with the TLB, since the cached virtual address mappings are no longer valid in the new virtual address space.

The simple solution is, when the page table BASE register (%cr3) is changed, meaning a switch to a different address space, the CPU hardware invalidates (flushes) all TLB entries, except those where the cached PTE has a special G (Global) flag set. (*we did not present this G flag in the conceptual model earlier in this unit*) The Linux kernel creates the PTEs for the shared kernel virtual address space so they all have the G flag. Therefore, when making a context switch, while the user-mode PTEs get flushed, all of the kernel-mode PTEs can remain in the TLB. This is a big performance boost. Still, when context switching from one process to another, we can expect TLB misses since the user-mode PTEs are never Global.

A better solution is to associate a unique integer ID with each virtual address space, and augment the key (tag) of the ATC/TLB cache so that it now consists of both the virtual page number AND the context number. Different architectures call this different things: **context number**, **Address Space ID**, **Process Context ID**: these all mean the same thing. This magic number is also stored in a CPU register, and is changed at the same time as the BASE address of the page tables.

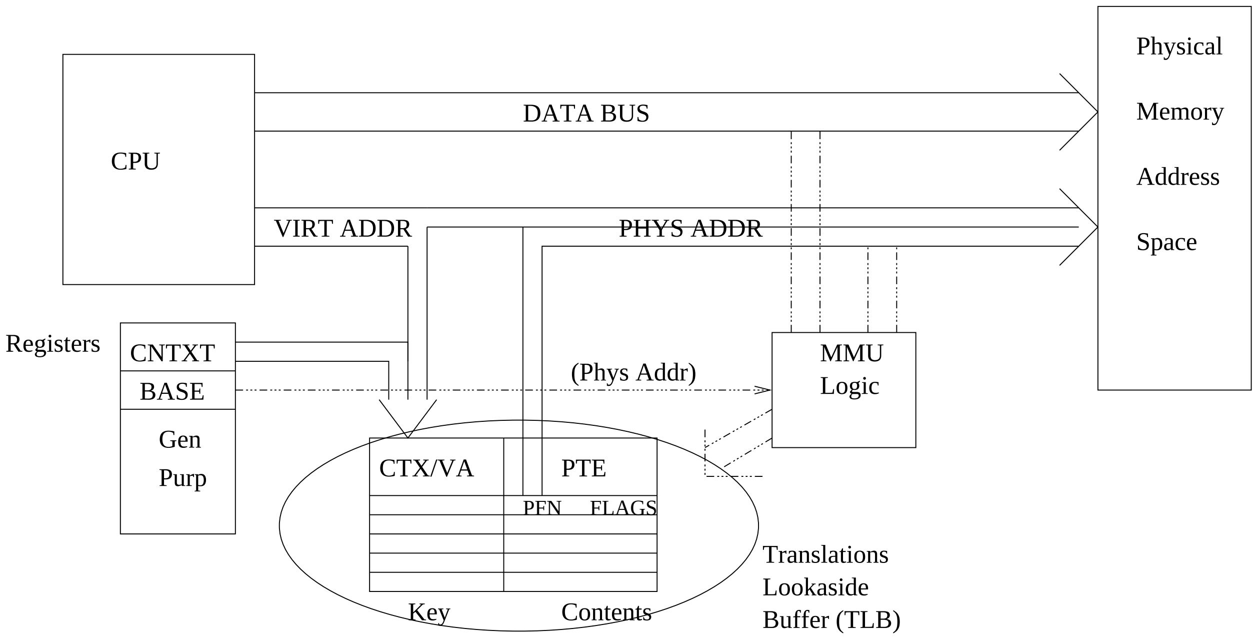
Most recent X86-64 processors have a PCID feature. The least significant 12 bits of the CR3 register contain the PCID. The remaining bits of CR3 contain the most significant bits of the physical address of the base of the page table (the PGD). This is possible because page directories and page tables are aligned on 4K physical address boundaries.

Another complication arises when the kernel has to change the PTE mapping for a given virtual page, e.g. an area of memory is being un-mapped, or changed from readonly to readwrite. If the PTE is cached in the TLB it must be flushed. On older versions of the X86 processors, it was necessary to flush the entire TLB, but modern versions allow specific entries to be flushed individually with a special (supervisor mode) instruction called `INVLPG`.

Even more fun with TLB flushing comes on multiprocessor systems and especially multi-threaded programs, because each processor maintains its own TLB. It could be that two or more physical processors are executing threads in the same address space (process). If the address space is changed in a way that would invalidate TLB entries, the other processors need to be alerted so they can flush their corresponding entries.

The TLB tends to be pretty small, on the order of 16 to 256 total entries. This corresponds to 64K to 2MB worth of address space that is being cached, and so is comparable to the performance of the L1 cache for the actual contents of memory. Because the TLB is small, performance is greatly improved when memory accesses are clustered so that, other than the first access, they "hit" the TLB cache. The kernel tries to optimize its internal data structures layout so as to keep frequently accessed data together within the same page. Thus the choice of memory address allocation within the kernel can have a profound impact on performance.

The figure below depicts the flow of address on a generic system with an MMU and a TLB, having the context number feature.



Aside: The X86 is a CISC (Complex Instruction Set) architecture in which the hardware *handles all aspects of ATC/TLB operations, including "walking" the page tables to load a PTE, and the eviction strategy for old TLB entries. On some RISC (Reduced Instruction Set) architectures, TLB/ATC management is handled by the kernel. A TLB miss causes a fault to be raised, and the kernel must determine the PTE and manually load it into the ATC/TLB.*

The UNIX user-level memory model

We have seen in Unit 3 that the virtual address space of a process consists of a number of regions. These include:
• text region: holds the executable code
• data region: initialized global variables
• bss region: uninitialized (0-filled) global variables including dynamically-allocated variables. The ending address of this region is known as the **break address**, and can be queried or set with the `brk` system call. The break address may turn out to not fall on a page boundary, in which case the bss region really extends to the next page boundary beyond the break address. The standard C library function `malloc` ultimately uses `brk` to request additional bss memory from the kernel. Another name for the dynamically allocated part of the bss region is the *heap*. Typically, the static part of the bss region (corresponding to the declared variables) is the first part (lower memory addresses), and the heap is contiguous with it.
• stack region: function call stack and local variables. With multi-threaded processes, each thread has its own stack. Stack regions have the unusual property that they automatically grow (in the direction in which stacks usually grow, towards low memory addresses on most architectures) without an explicit system call.
• dynamically-linked libraries code
• initialized global variables for dynamically-linked libraries
• uninitialized global variables for dynamically-linked libraries
• shared memory being used for inter-process communication
• memory-mapped files

We will now see that from the kernel's standpoint, these memory region names have no particular significance. The kernel views a virtual address space as a list of regions, each of which has associated with it:
• The starting virtual address of the region (always aligned to a page boundary)
• The length of the region (a multiple of the page size), or equivalently the ending virtual address of the region.
• Bitwise flags which describe properties of the region such as the protections, or the fact that the region automatically grows towards low memory
• Backing store: If the memory region is backed by a node in the filesystem, then this mapping is described by device#/inode# and offset. Otherwise, the region is **anonymous**, and has no corresponding file.

Tracking the Address Space in the Linux Kernel

As depicted in the diagram later in this unit, each `task_struct` in the kernel contains a pointer to a `struct mm_struct` which represents an entire virtual address space (think of "mm" as "memory map"). One of the many elements of `mm_struct` is `pgd`, the address of the PGD. There is also the head of a linked list of memory regions, which are kept in ascending order of starting virtual address, and each of which is described by a `struct vm_area_struct`:

```
 struct mm_struct {
         struct vm_area_struct *mmap; /* Head of linked list of regions */
         void *pgd;                   /* Addr of PGD */
         struct list_head mmlist;     /* Linkage to all mm_structs in system */
         int _file_rss,_anon_rss;     /* Current Resident Set Sizes */
         int total_vm;                /* Total VM size, in pages */
         int hiwater_rss;             /* High water mark of RSS usage */
         int hiwater_vm;              /* High water mark of total VM size */
 /* There’s plenty more to this structure that we’re skipping */
};

 struct vm_area_struct {
         struct mm_struct *vm_mm;             /* Pointer back to address space */
         struct vm_area_struct *vm_next;      /* Linked list pointer */
         void *vm_start;                      /* Start (virt) address of this region */
         void *vm_end;                        /* End address + 1 */
         unsigned long vm_flags;              /* VM_XXX flags */
         pgprot_t vm_page_prot;               /* PTE flags (see text about COW ) */
         struct file *vm_file;                /* Mapped file, or NULL */
         unsigned long vm_pgoff;              /* Offset (#pages) into mapped file */
         /* And there is more to this structure to be seen later */
};

/* Explanation of vm_flags (bitwise combinations )
VM_READ         Read permission
VM_WRITE        Write permission
VM_EXEC         Execute permission
VM_SHARED       Region may be shared among multiple processes
VM_GROWSDOWN    Region automatically grows towards lower addresses
VM_EXECUTABLE   Region maps to an executable file
VM_LOCKED       Region should be locked in memory and never swapped out
VM_IO           Special region for memory-mapped I/O
VM_DONTCOPY     Discard this region when forking instead of copying
VM_DONTEXPAND   Disallow expansion of region with mremap
VM_DENYWRITE    Disallow over-write of active text pages
*/
```

Establishing a new region with mmap

```
void *mmap(void *addr, size_t len, int prot, int flags,
                   int file_descriptor, off_t offset)
```

The mmap(2) system call causes the specified file which has already been opened with `file_descriptor` to be mapped into the process's address space, creating a new region. It can also be used to create a new anonymous memory region; these two choices are mutually exclusive. The first address of the mapped region will correspond to the specified `offset` in the file, which must be a multiple of the page size, for the specified length `len`. The virtual address at which to map the segment can be specified explicitly as `addr`, but usually it is left to the kernel to choose by passing NULL.

`prot` is a bitwise flag combination which can include the bits PROT_READ, PROT_WRITE or PROT_EXEC. These define the permissible types of memory accesses within the region. When mapping the region to a file, the requested memory access type must agree with the mode in which the file was open. E.g. PROT_WRITE will not be allowed if the file descriptor was opened with O_RDONLY.

`flags` is a bitwise flag which can contain, among others:
• MAP_SHARED: Write operations to mapped region change the contents of the file.
• MAP_PRIVATE: Mutually exclusive with MAP_SHARED: Write operations to the mapped region do not change the contents of the file.
• MAP_DENYWRITE: Setting it will prevent write access to the mapped file using the traditional write system call. See discussion at end of this section. While the Linux kernel currently prevents programs from setting this flag with the mmap system call, it is used internally by the kernel to protect executable files.
• MAP_GROWSDOWN: The region should have the property that it automagically grows towards low memory, i.e. it is intended to be a stack region.
• MAP_ANONYMOUS: A new, anonymous region is being requested. The file_descriptor and offset parameters are ignored.

Please read the man pages for mmap to obtain more complete information.

mmap within the kernel

The effect of (a successful) mmap call inside the kernel is to allocate a new `vm_area_struct` and to insert it into the singly-linked, ordered list of such memory regions that is hanging off the `mm_struct`. The `vm_flags` field is initialized based on the `flags` and `prot` arguments to the system call. E.g. PROT_READ maps to VM_READ, MAP_GROWSDOWN maps to VM_GROWSDOWN. If MAP_ANONYMOUS was specified, the `vm_file` is set to NULL, otherwise it points to the `struct file` (in kernel memory) that the file descriptor was referencing. Because the reference count (`f_count`) is incremented, the process may close the file descriptor after mmap, and the kernel "hangs on" to the open file instance. `vm_pgoff` is set to the `offset` argument, except the latter is specified in bytes, and is divided by 4096 (on X86 architecture) to arrive at vm_pgoff in pages. Accordingly, it is an error to specify an offset which is not page-aligned.

Shared vs Private Mappings

When MAP_SHARED is specified (equivalent to VM_SHARED in kernel), writing to the memory region is immediately and transparently visible to this or any other process through the traditional system calls such as `read`. Likewise, any changes to the file, e.g. through `write`, are immediately visible through the mapped region.

When the mapping is MAP_PRIVATE, Copy on Write (see below) happens. Upon the first memory write access to a page in a MAP_PRIVATE region, the association between that virtual page and the file is broken. Thus writes to a MAP_PRIVATE region do NOT cause the associated file to be modified. It is unspecified if writes to the file (e.g. via the write system call) which take place after the mmap association is made are then visible in the memory region, before the association has been broken by a write to the memory region. On the Linux kernel, they are (but see discussion of MAP_DENYWRITE).

Pages in a MAP_PRIVATE area, once they have been written to, become anonymous pages, because the association of that particular virtual page with that particular area of the mapped file has been broken. Therefore, if they need to be paged-out, they are sent to swap; they can not be paged-out to the original file.

File size not multiple of page size, file truncation, atime/mtime

The mapped region may extend beyond the current end of the file. This could happen either because the `len` parameter specified is larger than the file, or because after the mapping has been established, the file size is reduced (e.g. through the `truncate` system call). If the process attempts to access memory which corresponds to just beyond the current end of file, but not beyond a page size boundary, it will see bytes with a value of 0. However, when the process tries to go further out, beyond the page boundary, even though the memory address is within the bounds of the mapped region, the kernel will be unable to satisfy the page fault, because the backing store does not exist in the filesystem. Under this condition, the kernel must deliver a **SIGBUS** to the process.

Read and write access via a file-mapped region will set the inode atime and mtime fields appropriately, but not necessarily instantaneously, for reasons which should become clear later in this unit when we look at the PFRA.

Remapping, Unmapping

The `munmap` system call destroys a memory region and any associated mapping to a file. The `mremap` system call is used to make an existing memory region larger or smaller, or to move it to another virtual address. `mprotect` can change the protections (R/W/X) associated with a region (but those new protections must be consistent with how the file descriptor was open in the case of file-mapped regions). The reader is encouraged to read the man pages for these system calls carefully. How these system calls manipulate the `vm_area_struct`s should be obvious.

Exec and virtual memory

During the exec system call, a new address space is initialized and the existing address space of a process is freed. The page tables themselves are freed and any physical pages that the process had mapped are placed on the free list. It is as if the process had called `munmap` for each of its regions.

The basis of the initialization of the new address space is the filename passed to the exec system call. This refers to an executable file in one of the binary executable formats which that operating system and processor type can execute. The first few bytes of the file determines the executable format, and are known as the **magic number**. We have seen that the a.out file contains the text region image, the data region initial image (variable initializers), and the initial requested size of the bss region.

In Unit 3, a highly conceptual model was presented of how a new program is loaded into memory during exec. Now that we understand mappings between files and virtual memory, we can take a more refined look and see that executable "loading" is really the establishment of a number of mmap regions.

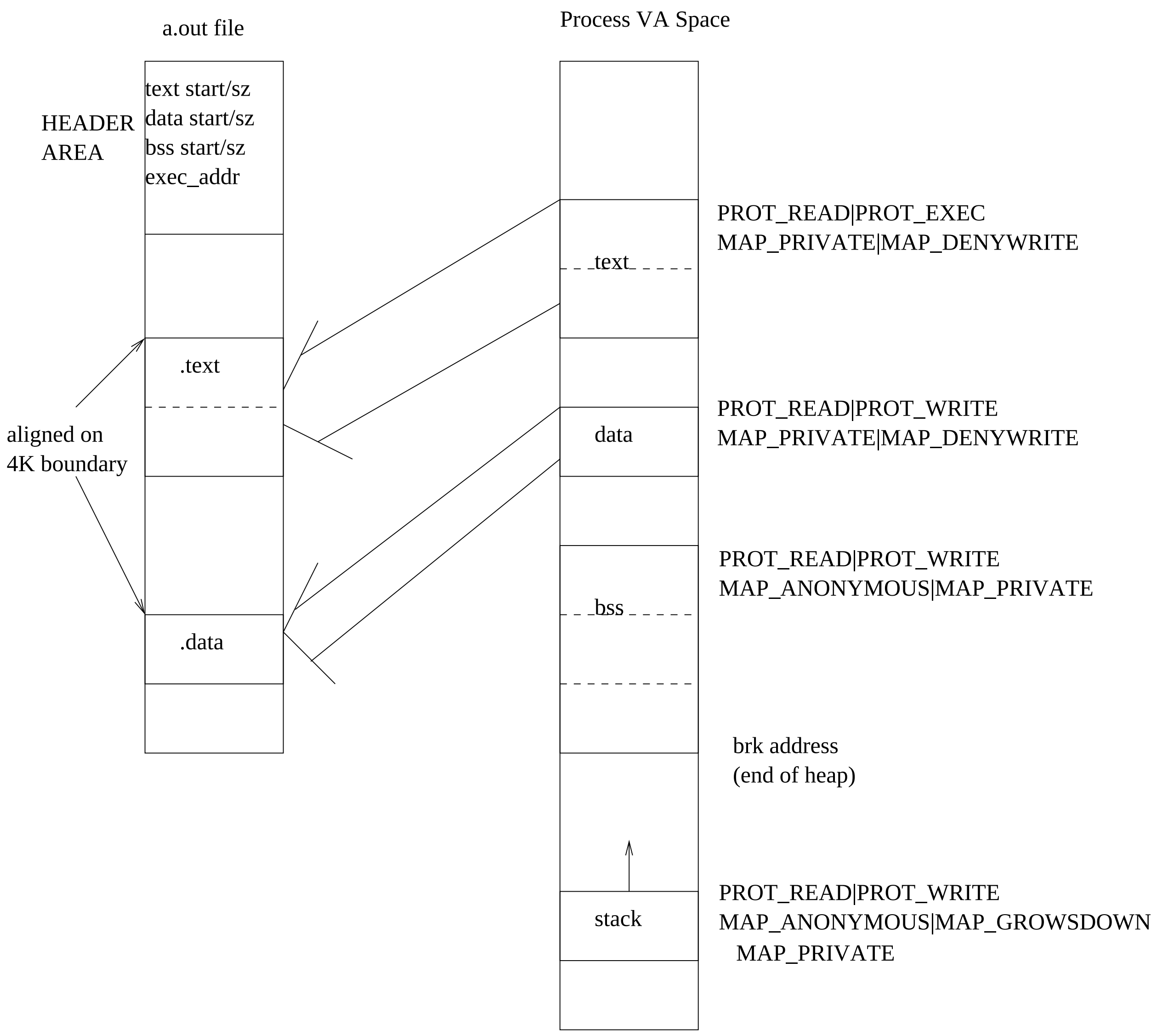
• The text region is created as if the process had used mmap to create a mapping to the area in the a.out file holding the text image. The protections of the text region are PROT_READ|PROT_EXEC. The flags are MAP_PRIVATE|MAP_DENYWRITE. Note that since page frames are never allocated to virtual address spaces until an actual access occurs, the very first thing which the new program does, upon attempting to fetch its initial opcode from memory and execute it, is to cause a page fault! The program is demand-paged in from the a.out file as needed.

• The data region is created as if by mmap with MAP_PRIVATE|MAP_DENYWRITE. The mapping is to the area of the a.out file which contains the .data section (image of initializer values). The protections are PROT_READ|PROT_WRITE. Because the mapping is MAP_PRIVATE, the program "sees" the correct initializers when they are first accessed, but writes to those variables don’t cause the unpleasant consequence of modifying the a.out file.

• The bss region is created, with the size requested in the a.out header, as a MAP_ANONYMOUS|MAP_PRIVATE mapping, with PROT_READ|PROT_WRITE protections. As it is accessed, the kernel will allocate page frames, zero-filling them first. As a MAP_ANONYMOUS region, the bss will be backed by swap space.

• The stack region is created with a small initial size and with the MAP_ANONYMOUS|MAP_PRIVATE|MAP_GROWSDOWN properties. The protections are PROT_READ|PROT_WRITE. The stack region will be grown as it is accessed, as described previously, with new page frames being zero-filled. The stack is also backed by swap space.

In the example below, a new program has been set up by exec with a text region of 2 pages, data region 1 page long, and an initial bss region of 3 pages.



Why the MAP_DENYWRITE flag?

Both text and data regions are mapped with the MAP_DENYWRITE flag set. The reason for this is the following situation: Let's say a program is running from a particular a.out file, and then someone comes along and re-writes that executable (e.g. it is being recompiled or reinstalled). However, the running program is still mapped to the file, so what gets paged-in would be a mixture of the original contents of the file and the newer contents. Clearly this won't work...it amounts to a corruption of the program. The DENYWRITE flag will prevent any kind of write access to the executable file, including truncation. It will fail with the error `ETXTBUSY`. However, one can unlink or rename the executing a.out file, and install a new version with the same name. Since the mapping is based on inode number, not pathname, the currently running process (or processes) will continue to use the old executable, while any new exec's would reference the new executable (since they evaluate a pathname, as if by **open**).

Those responsible for the development of the Linux kernel removed MAP_DENYWRITE from the list of flags that can be specified by the mmap system call (the flag is accepted but silently ignored). The reasoning went like this: User A which has read-only access to a file F could establish a read-only mmap mapping with DENYWRITE, even though they do not have write permission on F. This can create a denial-of-service attack, in that another user B, who does have write permission, would now be prevented from writing to the file. When the kernel sets up a new address space during exec, it is using internal versions of mmap, and is thus not actually using the flag MAP_DENYWRITE, but is specifying VM_DENYWRITE directly.

fork & clone interactions with Virtual Memory

We have seen that the `fork` system call is really a special case of the `clone` system call (on Linux) with the `clone_flags` set to all-zero. When the `CLONE_VM` clone flag is present, the child task will share all aspects of the virtual address space with the parent. This is accomplished within the kernel by having the `mm` field of the `task_struct` of the child simply be copied from that of the parent, so both tasks are pointing to and thus sharing the same `mm_struct`. This also means that there is one PGD for parent and child. Any system call affecting the virtual memory regions (`vm_area_struct`s) in the child has the same effect on the child and vice-versa.

However, in `fork`, the `CLONE_VM` is 0. This implies a deep copy of the `struct mm_struct` in the parent to form the memory map of the child. This means that each `vm_area_struct` is copied, forming a new linked list attached to the new `mm_struct` of the child. The page tables are also deep-copied. The child gets a new PGD which is a copy of the parent's, and any PUD, PMD and PTs are also recursively deep-copied. Each PTE in the parent is thus duplicated to the child, and if the page in question is Present, the reference count in the corresponding `struct page` (we'll cover this data structure shortly) is also incremented.

Regions that are writable and MAP_PRIVATE, including the data, bss and stack regions of any process, are set up for **copy-on-write**. This is described further in the section with that name.

The kernel also, when making copies of the page tables for the child, clears the PTE A(ccessed) bits in the child PTE's only. The parent is unaffected. This is because that virtual page in the child has not actually been accessed yet. The child is about to come to life and none of its virtual pages has been accessed!

The Page Frame Pool

Now that we have made an initial exploration of how the kernel manages the virtual address space of processes, let's take a look from the standpoint of physical memory. We'll then put it together and discuss the mechanisms for paging-in and paging-out.

A computer system has a certain amount of physical RAM. Some of that is taken up statically by the kernel, i.e. kernel text, data and bss, and system control structures that must exist at a certain physical address, such as the Interrupt Descriptor Table. There are additional demands for physical memory:
• Dynamically allocated kernel data structures, e.g. the task_struct, kernel-mode stacks, in-core inodes and directory entries, pending signals, loadable kernel modules, and shared kernel/hardware data structures such as page tables.
• User process address space. We've seen that this can be broken down into anonymous regions, which are not persistent, and file-mapped regions where resident pages correspond to specific portions of specific, persistent files.
• I/O buffers. In some cases the I/O buffer corresponds to a portion of a file, but in other cases (e.g. network packet buffers) it does not.
• Filesystem block caches: Filesystems sometimes need to cache disk blocks which are not the actual contents of a given file, e.g. directory entries.

The demand for page frames ebbs and flows throughout the life of the system. At certain times, there may be abundant page frames which are empty and uncommitted, and therefore available for immediate allocation. At other times, the page frame pool may be depleted, causing pressure for page frames to be freed up to satisfy new demand. The kernel, through the Page Frame Reclamation Algorithm (PFRA), always tries to keep a reasonable number of free page frames, so that an allocation request does not fail or encounter excessive delay. We'll study the PFRA shortly.

We will now consider how page frames are allocated by kernel routines, and how those page frames may be subdivided for efficient allocation of smaller objects.

Sidebar: Addressing Issues on X86-32

Recall that the kernel, in a 32-bit X86 environment, uses the last 1GB of virtual address space for itself, leaving the first 3GB to user-mode. Also recall that all kernel virtual addresses are shared, i.e. there is not a separate kernel virtual address space for each process or component of the kernel. Therefore, it would seem to be impossible for the kernel to have more than 1GB of internal data structures.

This is addressed through an interesting strategy. Linux further subdivides both physical and kernel virtual space so that the first 896MB of physical page frames map **directly** to the first 896MB of the last GB of virtual address space. This greatly simplifies kernel code as a physical page number can be used directly to compute the virtual address and vice versa. The last 128MB of the kernel's virtual address space is used to establish **temporary mappings** to physical pages which are above 896MB. The kernel calls these two areas of physical memory **zones**, the former being the NORMAL zone and the latter the HIGHMEM zone.

When allocating memory for internal data structures, the kernel uses the pool of pages in the NORMAL zone. Since user processes are not confined to the shared 1GB virtual address space, their pages always get pulled from the HIGHMEM zone first, and then, with reluctance, from the NORMAL zone if there is insufficient free memory in the HIGHMEM zone.

This klugery is not required for 64 bit architectures, where the kernel has ample virtual address space to make a direct mapping to all physical pages.

Page Descriptor Table

The kernel maintains a single, contiguous array of **page descriptor** structures, each of which represents one page frame:

```
struct page {
        unsigned long flags;                            /* bitwise flags */
        atomic_t _count;                        /* transient usage counter */
        atomic_t _mapcount;                     /* how many PTE mappings to this page */
        /* This is a GCC compiler extension known as a blind union */
        union {
                unsigned long private;                  // buddy order, et al
                struct address_space *mapping;          // reverse mapping
        };
        pgoff_t index;                          /* offset of page within mapping*/
        struct list_head lru;                   /* free/active lists */
        void *virtual;                          /* kernel virt address */
};
```

The page descriptor table, called `mem_map` (another term often used in UNIX kernels other than Linux is "core map") consumes a small amount of memory overhead. Specifically, on a 32-bit system, each `struct page` is 32 bytes long, so the overhead is 32/4096 or 0.7%. The page frame number is implicit by the index of the page descriptor within the `mem_map` array. For any given page, if it falls within the low-memory ("NORMAL") zone, the kernel can easily figure out the direct-mapped virtual address. Otherwise, if the page is currently mapped into kernel memory space, the `virtual` field says where.

A given page frame is either mapped (there is at least one PTE in somebody's virtual memory area, including the kernel's, that points to it) or it is free. To save space, some fields of the `struct page` are overloaded with mutually exclusive values. For example, the blind union `private / mapping` contains either information about the free page or information about the reverse mapping if it is mapped. The `_mapcount` field will be 0 if the page is free and >0 if the page is mapped somewhere. The confusingly named `_count` field is a temporary usage counter that is incremented when certain memory management routines are doing something with that page, and is used as a form of locking to prevent pages from disappearing while kernel code is playing with them. The `flags` field contains bitwise flags of the form `PG_xxxx` which encode various status values. For example, the `PG_locked` flag is set when an I/O operation is in progress on that page. The usage of the other fields in the `struct page` will be discussed during the rest of this unit.

Address space release (munmap)

When a `vm_area` is destroyed via the munmap operation, the kernel visits the page tables which are spanned by that region. For any PRESENT PTE, the PFN gets us to the struct page. The `mapcount` is decremented. If this was the last mapping to the page and the count is now 0, the page is moved to the free list. This process is recursive. If an entire page table worth of PTEs have been freed up, the page table itself can be returned to the page frame pool. Finally when the entire process address space is free, the PGD is freed.

If any PTEs indicate that this was an anonymous page that had been swapped out, the kernel updates its *swap map*, decrementing the reference count on that swap slot. When the reference count is 0, the swap slot is free to be re-used. (See later section on Paging Out)

The address space is also released upon process termination (via `exit` or via a signal), and when a new address space is going to be set up as part of `exec`. In these cases, the kernel calls an internal version of munmap for each vm_area_struct in that process's address space. Therefore it is never necessary for user code to explicitly free memory prior to program exit or exec!

Page Allocator -- Buddy System

The basic kernel page frame allocator is based on asking for one or more contiguous pages. Although most of the time kernel routines are asking for just one page, sometimes the kernel needs a rather large buffer which spans many pages, e.g. when setting up a Direct Memory Access transfer to/from an I/O device.

In general, memory allocation routines (e.g. `malloc()` in user mode) are faced with a tradeoff between space efficiency and time efficiency. If there is a request for say 7 contiguous "units" of memory, it is desirable in terms of space efficiency to satisfy that request with exactly 7 units. However, as memory is allocated and freed, **fragmentation** arises. There may be 3 units free, then 1 occupied unit, then 4 more free units. Even though there are 7 free units, they are not contiguous and thus do not satisfy the request.

In some allocation settings, e.g. on-disk filesystems, it is possible to de-fragment the system by swapping storage locations so as to group clumps of free units together. This is not possible in the C programming environment because once memory has been allocated, its raw address has been passed back as a pointer and there is no way to track down all references (including offset references) to that memory region, and thus no way to change the address of the region.

Given that fragmentation is unavoidable, the next best thing in terms of space efficiency is, given a request for N contiguous units, satisfy it with the smallest available contiguous region of size M units, M>=N. Now this leads to a time efficiency issue: how long will it take to search the list of available memory units to find this "best fit". In the case of the kernel, because memory allocation is a frequent and critical operation, it is desirable that this time be a constant one, not one which grows as the number of distinct free memory units increases because of fragmentation. Therefore, the Linux kernel makes a compromise, and adopts the well-known "buddy system" algorithm to give fairly decent space efficiency and lookup time which, although not strictly constant-time, has a small and fixed upper bound.

The "buddy system" algorithm maintains, for each of the memory allocation zones (e.g. NORMAL and HIGHMEM) an array of 11 pointers. The [0] element of this array points to a circular doubly-linked list of page descriptors for page frames which are free but not contiguous with other free pages. The [1] element points to such a list for groups of 2 contiguous pages. The [2] element does the same for groups of 4 contiguous pages, and so on up to the [10] element which collects groups of 1024 contiguous pages, i.e. 4MB. These lists are chained through the `lru` field of the page descriptor (which has other uses when the page frame is not free) and furthermore the `private` field of the first page descriptor in a contiguous chunk holds the order of that chunk. Subsequent page descriptors in the chunk are not part of the list but their existence is inferred by their position in the `mem_map`.

When a routine in the kernel wants say one page frame, the buddy system tries to take that from the [0] order list of free page frames. If there are none, it grabs a chunk of two page frames from the [1] list and splits it, handing one back to the caller and moving the other to the [0] list. If the [1] list is empty, this process continues moving up the ladder, e.g. taking a clump of 4 contiguous frames, returning 1 to the caller, and splitting the remainder into a single free frame and a pair of frames.

When a page frame or contiguous group of frames is de-allocated by the kernel to be placed back in the free pool, the "buddy system" checks to see if the freed page(s) is/are a contiguous "buddy" of another free page. If so, they are merged to form a higher-order contiguous region. This process can continue up to order 10, i.e. coalescing into a chunk of 1,024 free pages.

Note that the buddy system trades CPU time efficiency for absolute memory efficiency, by introducing an additional restriction. A group of pages on the, e.g. [2] list, i.e. a group of 4 contiguous pages, will always start with a page number divisible by 4. If say page frame numbers 1024-1027 inclusive are free, these can be used to satisfy a request for 4 contiguous pages. However, if 1025-1028 inclusive are free, this does not satisfy the request, because the start of the free area, 1025, is not a multiple of 4. Instead, the buddy system views this as a page of order 0 (#1025), two pages of order 1 (1026-1027) and another page (1028) of order 0. Likewise, a group of 8 contiguous pages must always start on an 8-page boundary, etc.

Given a page frame number `a`, its order-`N` buddy with page frame number `b` therefore satisfies the expression:
```
b==a ^ (1<<N)
```
Any page frame number `a` of order-`N` has a parent of order `N+1` which lives at:
```
p=a & ~ (1<<N)
```
These simple bit-arithmetic relationships allow the deallocation and conglomeration algorithm to be performed very quickly, without extensive data structures manipulation and resulting memory accesses. The penalty is slightly increased fragmentation.

Slab Allocator

In general-purpose user-level C code, `malloc` is the primary tool for dynamically allocating heap memory. Allocations tend to be for objects of varying sizes. With every chunk of memory returned to the caller by `malloc`, there are generally at least two additional hidden variables which live just before that chunk, and which contain the size of the chunk, and a pointer to other chunks. However, in the kernel, there is no `malloc`! A lot of the dynamic allocation activity in the kernel is characterized by requests for small, fixed-size objects, such as `task_struct`, `struct file`, inodes, etc.

To improve both spatial and temporal efficiency, the Linux kernel uses a system called the **Slab Allocator** for such small, fixed-size objects. This system is fairly complicated and will not be fully described here. The kernel grabs page frames and uses them as slabs, storing an internal descriptor at the beginning of the slab to track how the page frame or frames is being subdivided. Because the kernel is a tightly-controlled piece of code and the type of data structures which will need to be allocated and released frequently is understood in advance, the slab allocator can be optimized so that it can quickly allocate and free objects of a given type.

When the kernel requires space for a new object of a certain type, it consults the slab associated with that type. The slab descriptor bitmap quickly determines if there is a free slot open, and if so where. If there are no free slots, the slab allocator goes back to the page frame allocator and asks for an additional page (not necessarily contiguous) which is subdivided into slots.

Likewise, de-allocation of an object allocated through the slab allocator means simply noting in the descriptor that the slot is free. If all slots in the page frame are free, the page frame itself is released to the free pool.

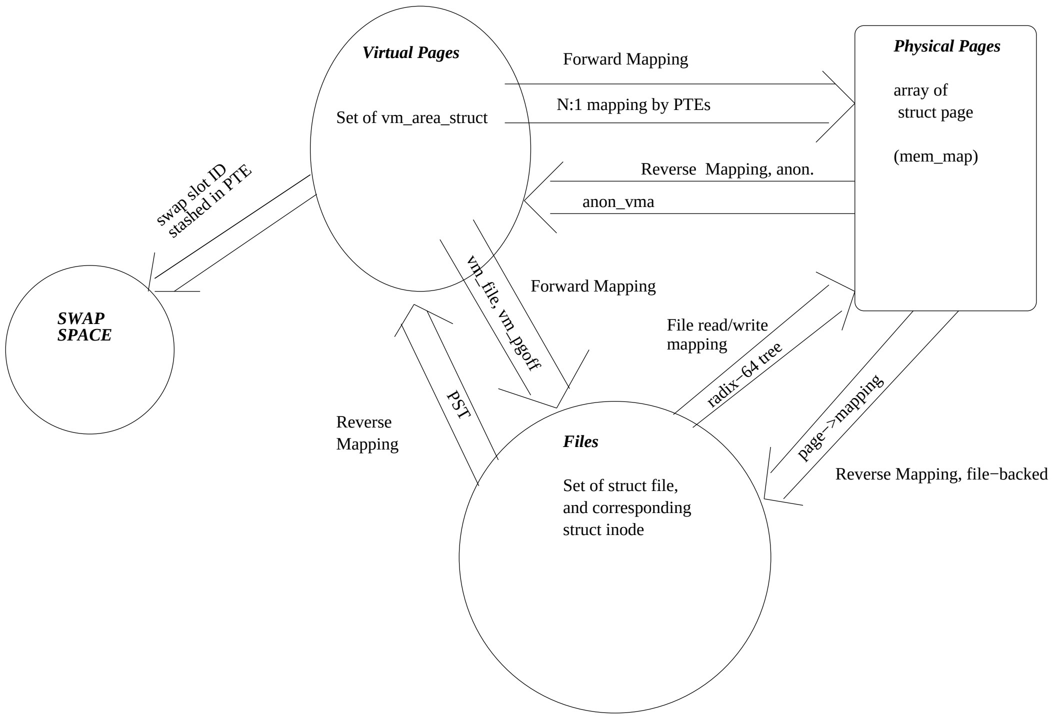
The slab allocator is both space efficient (the overhead is just 1 bit in the bitmap per object, plus a few bytes for the overall slab header) and time efficient (no convoluted data structures). This is important because these small data structures are frequently being allocated and released by the kernel.

The Trinity of Memory and Addressing

We can look at randomly-addressable things from three standpoints:
• Virtual pages: The set of all virtual memory regions in all address spaces along with their associated properties
• Physical pages (page frames)
• The file and disk I/O system. Each file is randomly-addressable and portions of the file may be cached in physical memory. In addition, a file or a raw disk partition may be used as **swap space** to hold anonymous virtual pages when they have to be paged-out.

There are relationships in both directions between each of these three views, which are managed by various kernel data structures. The diagram near the end of this unit shows all of these together and can be quite intimidating. The relationships, or mappings, are as follows:

• **Forward Mapping**: Given a virtual page (i.e. a specific virtual memory area and offset within that area), determine either 1) what page frame holds it or, 2) if no page frame holds it, how do we find the canonical contents of that page frame? The MMU's page table data structure, which ultimately delivers us to the PTE, determines the page frame. This is an N:1 mapping because a given page frame can be shared among multiple virtual memory regions (e.g. MAP_SHARED mmap).
• **Reverse Mapping**: For a given page frame, find all PTEs that map to it. Different data structures are used for reverse mapping depending on whether the given page is anonymous or file-mapped.
• **File read/write (I/O) Mapping**: When a given page-sized chunk of a file is wanted for a read or write system call, determine if that chunk is already cached in memory and if so, in what page frame?



These mappings form a circle, and thus it is difficult to present this material which has circularities in our comprehension prerequisites. We must jump into it at some point, so we will start with Forward Mapping!

Page Fault Handling -- Forward Mapping

The page fault exception is used by the CPU to indicate an access to an invalid or un-translated virtual address. However, this is usually not a real problem. The kernel enjoys handling page faults, and uses page faults to implement a number of features, one of which is **Demand Paging**: page frames are not actually allocated, and I/O operations (if applicable) are not actually initiated, until a process establishes the need to do so by attempting to access the corresponding virtual page. This is a significant speed and memory usage improvement. Without demand paging, programs would take a long time to load and start, because the entire program code (including all those libraries) and data would have to be read in from disk. With Demand Paging, only the code and data which is actually used consumes resources.

The kernel’s page fault handler knows:

- The identity of the faulting process (`current` global variable)
- The attempted virtual address.
- The type of access (read/write).
- Whether the virtual page in question had a valid PTE
- The registers at the time of fault (including user/supervisor flag).

The page fault handler can have one of two general outcomes: Either the page fault is resolved, resulting in the establishment of a valid PTE (which has the appropriate R/W/X permissions for the access being attempted) pointing to a valid page frame with the correct data in it, or the page fault can not be resolved, resulting in the delivery of a SIGSEGV, SIGBUS or SIGKILL to the process. Let’s take a look at the page fault handler code:

```
/* Please note: All of this code is fictional, but based on a true story.
 * It is a heavily simplified and cleansed condensation of actual Linux
 * kernel source code for page fault handling.  Many of the variable and
 * function names have been changed to protect the innocent.  The interested
 * reader is invited to begin at /usr/src/linux/mm/memory.c           */

/* handle_pagefault_usermode() is the fictitious name of the fault handler
 * which is invoked when the MMU gives a page fault and the kernel determines
 * that the fault happened while in user mode.  Because this is a synchronous
 * entry into the kernel, the current variable is that of the process which
 * caused the page fault.  Upon return from this function, we will return to
 * user mode and either the faulted instruction will be re-started or a
 * signal handler will be invoked                                      */
handle_pagefault_usermode(void *addr, int access_type)
{
 struct vm_area_struct *vma,*growsdown;
 void *new_addr;


          if (addr >= KERNEL_MEMORY_START)      // Attempt to access kmem from user
                    force_sig(current,SIGSEGV);
          /* Note: the force_sig function posts the requested signal to the
           * specified task.  If that signal is currently being ignored or
           * blocked, the setting is changed to SIG_DFL which usually terminates.
           * We can’t ignore or defer the signal because the current instruction
           * has faulted and therefore continued execution makes no sense */

          growsdown=NULL;
          /* Is the fault address inside of a defined region? */
          for(vma=current->mm->mmap;vma;vma=vma->vm_next)
          {
                    if (addr>=vm->vm_start && addr<vm->vm_end) break;
                    if (vm->vm_flags & VM_GROWSDOWN &&
                              vm->vm_start-addr < STACK_FUZZINESS &&
                              regs->esp <= addr + STACK_FUZZINESS)
                                        growsdown=vma;
          }
          if (!vma && growsdown)
          {
                    new_addr= addr&˜(PAGE_SIZE-1);        // round down to page boundary
                    vma=growsdown;
                    /* There are additional checks, e.g. will the newly grown
                     * stack exceed resource limits, or collide with another area */
                    vma->vm_start=new_addr;
          }
```

```
         if (!vma)          // No valid region found
                  return force_sig(current,SIGSEGV);

         /* Inside of a region, may have been a protections fault */
         if (!protections_ok(vm->vm_flags,access_type))
                  return force_sig(current,SIGSEGV);
         /* We have now proved that the user’s access is valid,
          *       let’s resolve the page fault */
 try_again:
         vp_addr = addr & ˜(PAGE_SIZE-1);   // Start of virtual page w/ fault
         switch (handle_mm_fault(current->mm,vma,vp_addr,access_type))
         {
           case VM_FAULT_OOM:
                  if (current->pid == 1) {      // Oh no, can’t kill init!
                           yield();             // Let PFRA work
                           goto try_again;
                  }
                  printk("VM: Killing process %d0,current->pid);
                  do_exit(SIGKILL);
                  /*NOTREACHED*/
           case VM_FAULT_SIGBUS:
                  force_sig(current,SIGBUS);
                  break;
           case VM_FAULT_MAJOR:
                  current->maj_flt++;
                  break;
           case VM_FAULT_MINOR:
                  current->min_flt++;
                  break;
         }
}

handle_mm_fault(struct mm_struct *mm,struct vm_area_struct *vma,
                                        void *addr,int access_type)
{
 pte_t **pgd,**pud,**pmd,**pt;
 pte_t *ptep,pte;
         pgd=mm->pgd;
         /* PGD is always allocated.  Make sure PUD/PMD/PT are allocated */
         /* corresponding to the faulted address.  The xxx_alloc functions */
         /* return a pointer to the next table, possibly allocating a new */
         /* page frame for the table and installing a pointer to it in the */
         /* enclosing table.  Physical addresses are mapped by the kernel */
         /* to virtual addresses for this temporary purpose; this mechanism */
         /* is very arch-specific and beyond the scope of this class */

         if (!(pud=pud_alloc(mm,pgd,addr))) return VM_FAULT_OOM;
         if (!(pmd=pmd_alloc(mm,pud,addr))) return VM_FAULT_OOM;
         if (!(pt=pt_alloc(mm,pmd,addr))) return VM_FAULT_OOM;
         /* Compute offset of PTE within Page Table */
         /* On X86-32, PAGE_SHIFT is 12 bits and PAGE_MASK is 0x000003FF */
         /* On X86-64, PAGE_MASK becomes 0x000001FF */
         ptep= &pt[((unsigned long)addr >> PAGE_SHIFT) & PAGE_MASK];
```

```
          pte= *ptep;

          if (!pte_present(pte))                // PRESENT bit is 0
          {
                    if (pte == 0)               // First time access or file-backed
                    {
                              if (vma->vm_file)  // File-mapped
                                        return pagein_from_file(mm,vma,addr,ptep,access_type);
                              else                          // Anonymous
                                        return pagein_anon(mm,vma,addr,ptep,access_type);
                    }
                    /* PTE encodes swap loc for paged-out anon page */
                    return pagein_from_swap(mm,vma,addr,pte,ptep,access_type);
          }
          /* PTE is present, must be a COW protection fault */
          if (write_attempt(access_type))
          {
                    pfn_t pfn;
                    pfn=((unsigned long)pte >> PFN_SHIFT) & PFN_MASK;
                    /* This next condition is common after fork-exec */
                    if (mem_map[pfn]._mapcount==1)
                    {
                              /* Some code elided to check if MAP_PRIVATE page
                               *        that had been marked down for COW but had
                               *        only a single mapping, needs to adjust
                               *        the radix-64 tree to invalidate           */

                              *ptep|=PTE_WRITE;
                              return VM_FAULT_MINOR;
                    }
                    void *new_pf;
                    /* Allocate a new page frame and provide a valid temporary */
                    /* virtual address so we can access it */
                    if (!(new_pf=pageframe_alloc(1))) return VM_FAULT_OOM;
                    memcpy(map_pf_to_va(new_pf),addr,PAGE_SIZE);
                    *ptep|=PTE_WRITE;
                    return VM_FAULT_MINOR;
          }
}

/* This function is only called for first-time access to an anon virt page */
pagein_anon(struct mm_struct *mm,struct vm_area_struct *vma,void *addr,
                    pte_t *pte_p,int access_type)
{
  void *new_pf;
  pfn_t pfn;                    // 32 bit or 64 bit
  unsigned pte_bits;
          pte_bits=vma->vm_page_prot | PTE_PRESENT;
          if (read_attempt(access_type))
          {
                    /* For first-time read, we can grab a shared COW 0 page */
                    new_pf=global_shared_zeropage_paddr;
                    if (!new_pf) return VM_FAULT_OOM;
```

```
                  pte_bits &= ˜PTE_WRITE;      // Not writable -- COW
         }
         else
         {
                  new_pf=pageframe_alloc(1);
                  /* map_pf_to_va provides a (temporary) virt addr */
                  bzero(map_pf_to_va(new_pf),PAGE_SIZE);
         }
         pfn=(unsigned long)new_pf>>PAGE_SHIFT;
         mem_map[pfn]._mapcount++;    // Update core map
         /* Fill in the PTE.  PFN_SHIFT is the bit pos of the PFN in the PTE */
         *ptep= (pfn<<PFN_SHIFT) | pte_bits;
         update_rss(ANON);  // Update rss, highwater, etc. stats.
         return VM_FAULT_MINOR;
}

pagein_from_file(struct mm_struct *mm,struct vm_area_struct *vma,void *addr,
                  pte_t *pte_p,int access_type)
{
 void *new_pf;
 struct page *new_pf;
 pfn_t pfn;
 unsigned long pte_bits;
 int rc;
         /* Determine offset within file, in units of pages */
         va_offset=(unsigned long)(addr-vma->vm_start) >> PAGE_SHIFT;
         file_offset=va_offset+vma->vm_pgoff;
         /* See if that page is already resident in memory */
         rc=VM_FAULT_MINOR;
         if (!(new_pf=file_address_space_search(vma->vm_file,file_offset)))
         {
                  /* No, need to allocate a fresh page frame */
                  if (!(new_pf=pageframe_alloc(1))) return VM_FAULT_OOM;
                  pfn=(unsigned long)new_pf >> PAGE_SHIFT;
                  new_pf= &mem_map[pfn];
                  /* Ask the filesystem module to initiate a disk read
                   * operation, depositing the result in physical page new_pf.
                   * This puts the task into a non-interruptible SLEEPING state
                   * until the I/O operation completes (or fails           */
                  if (!do_file_readpage(vma->vm_file,file_offset,new_pf))
                  {
                           pageframe_free(new_pf);
                           return VM_FAULT_SIGBUS;
                  }
                  rc=VM_FAULT_MAJOR;
         }
         pfn= new_pf-mem_map;          // PFN is implicit by index within array
         new_pf->_mapcount++;          // Here comes new PTE
         pte_bits=vma->vm_page_prot|PTE_PRESENT;
         *ptep=pfn<<PFN_SHIFT|pte_bits;
         update_rss(FILE);
         return rc;
}
```

```
pagein_from_swap(struct mm_struct *mm,struct vm_area_struct *vma,void *addr,
                   pte_t pte,pte_t *pte_p,int access_type)
{
 void *new_pf;
 pfn_t pfn;
 int pte_bits;
 unsigned long swap_area,swap_slot;
          swap_area=(pte&SWAP_AREA_MASK)>>SWAP_AREA_SHIFT;
          swap_slot=(pte&SWAP_SLOT_MASK)>>SWAP_SLOT_SHIFT;
          if (!(new_pf=pageframe_alloc(1))) return VM_FAULT_OOM;
          /* This operation results in a non-interruptible SLEEP */
          if (!read_from_swap(swap_area,swap_slot,new_pf))
          {
                   pageframe_free(new_pf);
                   return VM_FAULT_SIGBUS;
          }
          /* Some more complexity elided to deal with a SHARED, ANON
           * page that had more than one mapping and had been swapped out */
          mark_swap_slot_free(swap_area,swap_slot);
          pfn=(unsigned long)new_pf >> PFN_SHIFT;
          pte_bits=vma->vm_page_prot|PTE_PRESENT;
          *ptep=pfn<<PFN_SHIFT|pte_bits;
          mem_map[pfn]._mapcount++;
          update_rss(ANON);
          return VM_FAULT_MAJOR;
}
```

Prior to the code seen above, the kernel examines the faulting address and the user/supervisor mode bit. If the faulting address is inside the kernel's reserved virtual address range AND we were in user mode, then a SIGSEGV is sent to the process, no questions asked. If we were in supervisor mode, then the page fault happened inside kernel code. The kernel tries to avoid any situations where this happens, as the code for handling this is messy. We won't be considering kernel-mode page faults in the Linux kernel any further.

Now that we have determined that it was a user-mode program that got us into the page fault handler, and that the access was not to the kernel reserved address space, we have to see if this was a legitimate virtual memory access. This is done by examining the virtual address space description contained in the list of `vm_area_struct`s to see if the faulting address is inside of an existing valid virtual memory region. If it is, we then need to determine if there has been a protection violation (access attempt mode doesn't match PTE mode). There are two exceptions to this: stacks and copy-on-write.

Stacks

When the stack grows e.g. through a push instruction, the stack pointer is decremented first and then the write takes place. If the stack pointer is now lower than the current start of the stack region, a fault will be incurred. If the faulted address is "close" to the stack pointer (an allowance is made for certain X86 instructions which access memory on the stack first, then adjust the stack pointer) and if the memory region just above the faulted address has the GROWSDOWN property, then this stack region is expanded by one page toward low memory addresses ("down") as if by the `mremap` system call. There are checks performed to make sure this doesn't exceed the stack size ulimit resource limit, or cause the stack to collide with another region. This automagical stack growth is quite essential to transparent program execution.

This adjustment of the `vm_start` address does not by itself resolve the page fault. The logic flow now continues, the same as a page fault in any other valid memory region.

Copy-On-Write (COW)

There are several cases where the kernel must logically make a copy of a page. One common example is during `fork()`. Another is when a file is mapped MAP_PRIVATE, a page is initally faulted in for reading from the file, and then it is written to, requiring that the association with that part of the file be broken.

If the kernel actually allocated new page frames and copied, byte-for-byte, the entire contents of each page frame of the region being copied, that would be a terrible performance penalty. Instead, the kernel continues to point both virtual mappings to the same page frame, but marks down the PTE entries in question to disallow write access. In the case of a fork, the W bit is turned off in both the original parent PTE and the new child PTE. The `vm_page_prot` field of the `vm_area_struct` is modified so that any new PTE mappings in the shared memory regions (which would otherwise have the PTE_WRITE bit turned on because they have VM_WRITE) caused by a page fault are installed with the W bit off, which will allow the page frame to be shared until somebody tries to write to it.

Likewise in the case of a MAP_PRIVATE file mapping, the `vm_page_prot` of the associated region will never have the PTE_WRITE bit on. Pages which are faulted in by reading are installed with read-only PTEs. If the process later tries to write via the memory mapping, this causes a page fault which alerts the kernel to break the mapping for that page and make a private copy. Both of these COW examples have the same implementation and effect even though the purposes differ.

Regions that have the VM_SHARED property (i.e. MAP_SHARED memory mapped regions) are **not** copy-on-write since by definition they are shared after a fork. The PTEs are not modified and the parent and child will continue to map to the same page frames for all pages in this shared region.

As long as none of the processes with the C-O-W mapping attempts to write, there is no need to do anything further. They will each see the correct data which are unchanged from the point of forking, in the first example case, or from the original contents of the file, in the second case. As soon as any process tries to write, it will incur a page/protection fault. The kernel recognizes this COW situation because the faulting address is valid, the virtual memory area flags say we should be allowed to write to this area, AND there was a PTE PRESENT for that address.

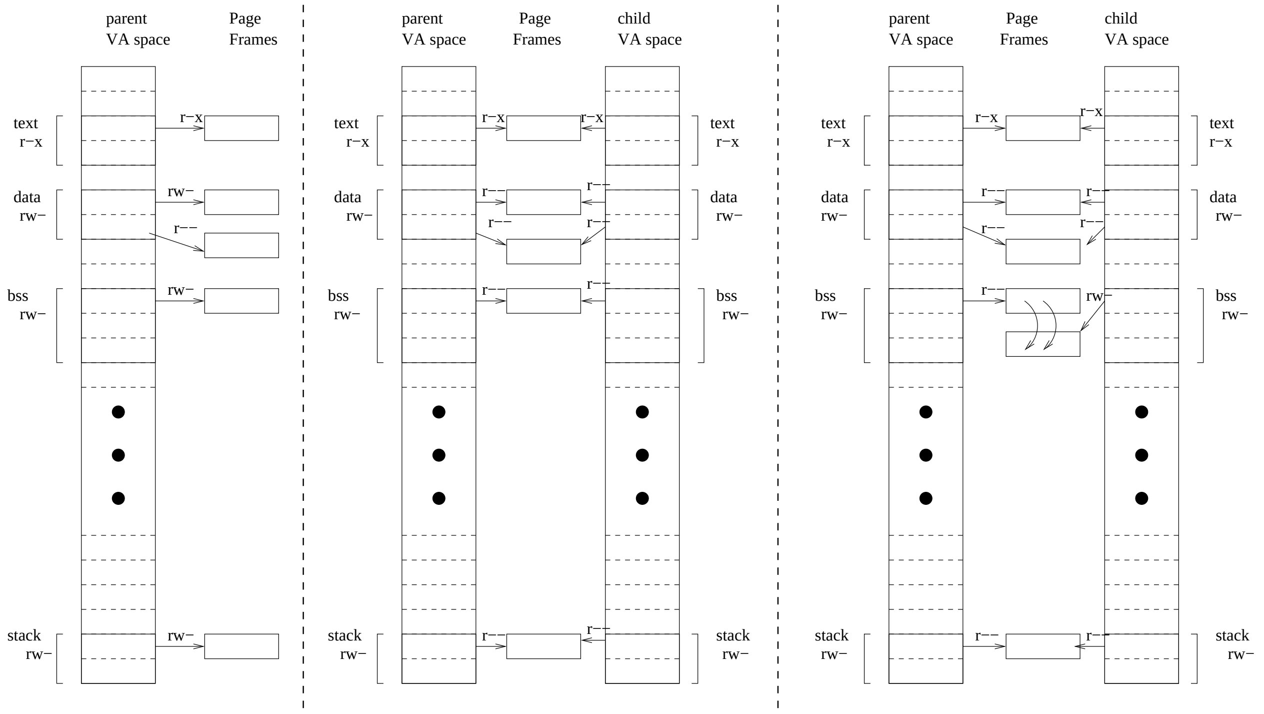
To resolve the COW fault, the kernel grabs a new free page frame, copies the contents of the existing (shared) page frame to the new frame, and then updates the PTE of the faulting process to point to the new frame, turning on the PTE Write Enable bit at the same time. The other process(es) sharing that mapping continue to see it read-only.

One important optimization on the above happens when a process forks and then either the parent or the child (but usually the child in the common UNIX programming idiom) execs. Then the shared memory region is unmapped from one of the processes. As we have seen, the kernel maintains a core map data structure giving the state of each page frame, and can track how many references there are to any given page frame. So the unmapping results in the reference counts of the shared pages going back down to 1. When the kernel sees this condition during page fault resolution, it does not have to allocate a new page frame and memcpy. It merely changes the PTE to turn the write enable bit back on. The same thing occurs if a shared page is COW'd by one process, say the child, and then the other process faults on the same page. The COW resolution had reduced the reference count to 1.

Note that the MAP_PRIVATE mapping of a file creates a hybrid region. Within that region, some virtual pages may be mapped in read-only, and still correspond to the original file, while others that have been dirtied are effectively anonymous pages. They will have to be written back to *swap* which we will define shortly, if their page frames get reclaimed. We'll see how the `mapping` field of the page descriptor is used to distinguish, on a page frame by page frame basis, whether a given page frame is still file-backed or is anonymous.

There is another example of copy-on-write: if we have an anonymous region and a new page is demand-paged via a read mode page fault. We need to see a page full of 0 bytes. But if we are only reading it, why do we need a distinct page frame full of 0s for each such "virgin" virtual anonymous page? We don't! The kernel keeps a single page frame full of 0s and shares it, using the C-O-W mechanism to split the sharing when anyone tries to write to it.

In the illustration below, before the fork, the parent process has demand-paged one page of text, one page of bss, one page of stack, and two pages of data. It wrote to the first data page but the second page has only been read, so its PTE is still r--. After the fork (second panel), all PTE W bits are off. In the third panel, the child attempts to write to the bss page. This causes copy-on-write. Note that the parent's PTE is not affected.



Invalid Addresses and SIGSEGV

If, however, the faulted address is outside of all regions and is not associated with stack growth, or if the attempted access is a violation of the regions protections, a signal `SIGSEGV` is forcibly posted to the current process. "Forcibly" means the kernel doesn't care if the process has this signal number blocked or ignored: The user program attempted to access an illegal virtual memory address, and there is no point in letting it continue on its current path. The address will not magically become valid, and therefore the faulted instruction can never be satisfied and would loop forever. The signal then terminates the process.

If the signal has a handler, but the signal is currently being blocked, the signal is unblocked, the handler is reset to SIG_DFL, and once again the signal kills the process. The reasoning is that a handler should not be invoked if the corresponding signal is being blocked.

It is, however, permissible for the signal to be *handled* as that will force execution to jump to another, presumably less flawed part of the program. This technique is often used in complex, modular applications (such as the open-source image editor *The Gimp*) to isolate a flawed module and prevent it from crashing the entire program and possibly leading to a loss of data.

Benign Page Faults / Paging-in

Page faults which are within an existing region and which are not the result of protection violations are therefore not the result of any program error, and need to be resolved by establishing a valid forward mapping PTE that is pointing to a valid physical page frame which contains the correct contents corresponding to the virtual page. This page fault resolution mechanism can have one of four mutually exclusive outcomes:

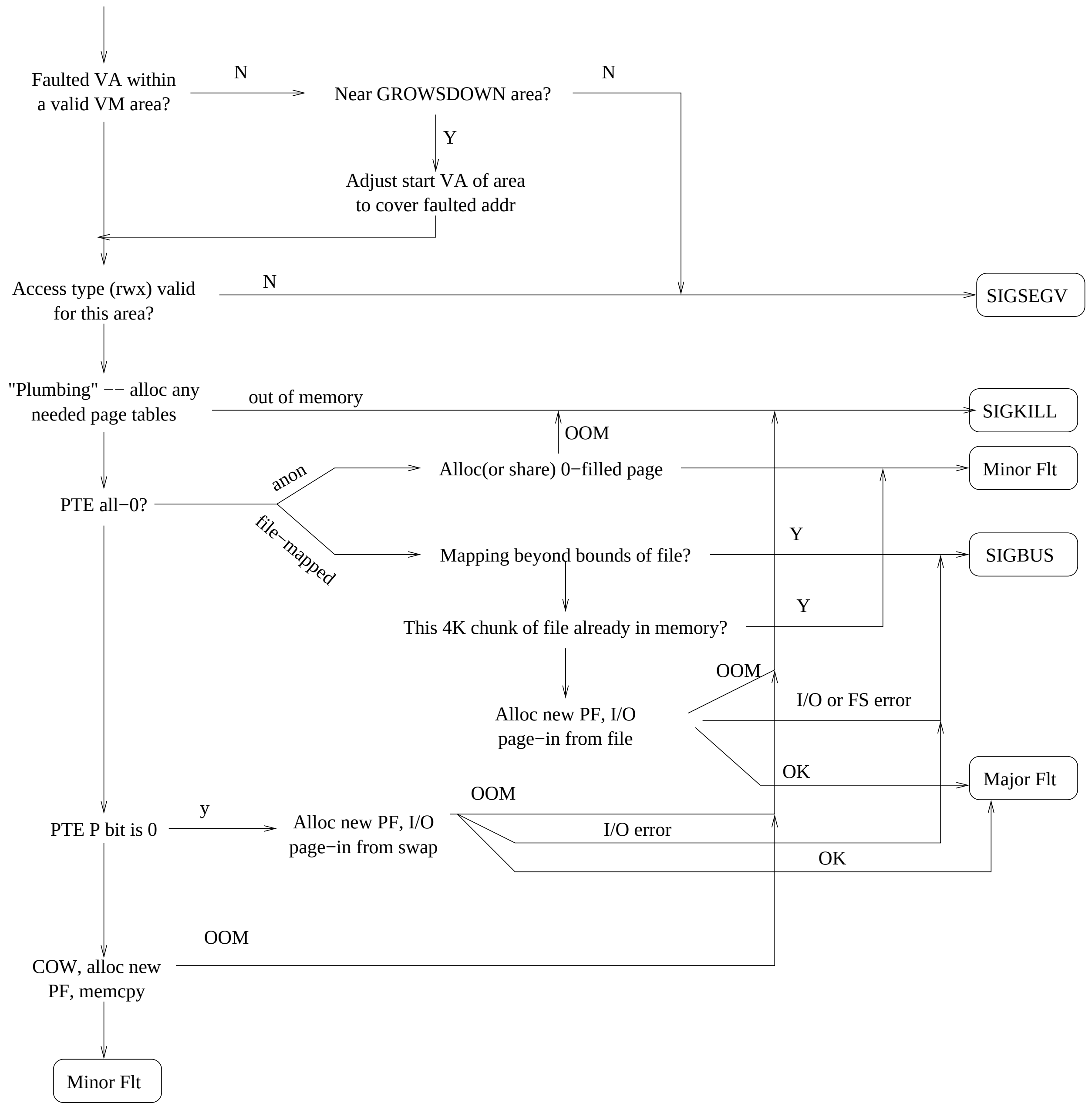
• `Minor fault`: The page fault is resolved quickly without requiring a disk I/O operation or putting the process to sleep. E.g. demand-paging new bss pages (just allocate a page, zero fill and return), paging-in file-mapped pages which are still cached in memory, or breaking up a Copy-on-Write shared page frame. Minor faults are good.

• `Major fault`: It was necessary to sleep pending an I/O operation to satisfy this fault. Major faults are not necessarily evil (e.g. demand paging an executable file from disk) but we would like to minimize their number.

• `Bus Error`: The kernel was unable to page-in the page because of an error other than being out of memory. E.g. the page is mapped to a disk file but an I/O error occurred trying to access the disk, or the mapping is no longer valid because the file has been truncated. A SIGBUS will be delivered to the process.

• `Out of Memory`: The kernel ran out of memory to allocate a free page frame or an internal data structure. This is very bad. Under ordinary circumstances, the kernel tries to prevent this from happening by maintaining a pool of free memory. If this does happen, the kernel has no choice but to terminate the current process. See also discussion below on "over-commit".

The Linux page fault handling algorithm is summarized by the following simplified flowchart:


Faulted VA within a valid VM area?
N
Near GROWSDOWN area?
N
Y
Adjust start VA of area to cover faulted addr
Access type (rwx) valid for this area?
N
SIGSEGV
"Plumbing" −− alloc any needed page tables
out of memory
SIGKILL
OOM
anon
Alloc(or share) 0−filled page
Minor Flt
PTE all−0?
file−mapped
Mapping beyond bounds of file?
Y
SIGBUS
This 4K chunk of file already in memory?
Y
OOM
Alloc new PF, I/O page−in from file
I/O or FS error
OK
Major Flt
PTE P bit is 0
y
Alloc new PF, I/O page−in from swap
OOM
I/O error
OK
COW, alloc new PF, memcpy
OOM
Minor Flt

On-demand creation of Page Tables

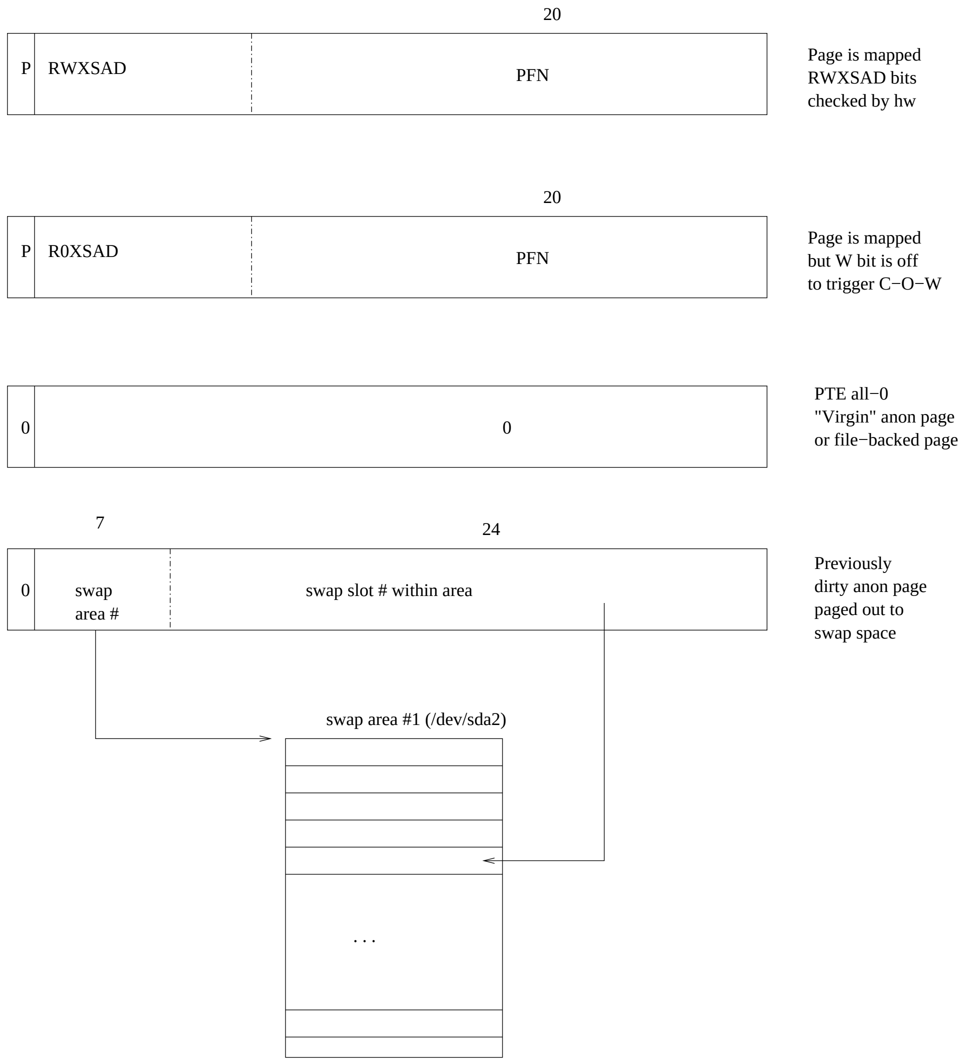
Recall that the structure of the page table is multi-level and sparse. Only the top-level Page Global Directory is pre-allocated. The kernel uses **lazy allocation** in creating the page upper directories, page middle directories, and page tables. Therefore, at this time of page fault resolution, the kernel must (recursively) allocate the directories/tables mapping the faulted virtual memory address, if they are not already allocated. When new page tables or page directories are allocated, they are set to 0, meaning none of the entries are viewed by the hardware as PRESENT yet. The kernel then installs new Page Directory or Page Table Entries as needed.

Paging-in from where?

To complete page fault resolution, the kernel must locate the canonical copy of the data corresponding to the virtual page which is faulted. In brief, this can fall into one of the following cases:
• The virtual page is file-mapped.
• The virtual page is anonymous and not previously written to.
• The virtual page is anonymous and previously dirtied, and swapped out.
• The virtual page is already present in physical memory, and this is a copy-on-write.

The kernel can determine which case applies by looking at the bits of the PTE corresponding to the faulted address, and by looking at the attributes of the corresponding `vm_area_struct`.
• The PFN field is all zero and the Present bit is 0, and the `vm_area_struct` indicates that this region is mapped to a file. The corresponding page-sized chunk of the file might already be cached in a page frame because of recent read/write sytsem call activity, or it may require the allocation of a new page frame and the initiation of a disk I/O operation. See "File Mappings"
• The PFN field and Present bit are all 0 and the `vm_area_struct` tells us that this is an anonymous region. The PTE being entirely 0 tells us that this virtual page is "virgin": it has not previously been written to and swapped out. This situation is always resolvable as a minor fault: The kernel finds a page frame from the free pool and maps it to the faulted virtual page. If the faulted access was a write, the kernel 0-fills the page. However, for a read access, the kernel optimizes by mapping to a single, shared physical page containing all 0, and setting up a COW situation by making the PTE protection bits read-only. At a later time, if the process writes to this page, things will be straightened out by Copy-On-Write. This behavior of **zero-filling** new anonymous pages is very important for security. Otherwise, sensitive information from another process could be viewed when the page frame is recycled.
• The PFN field of the PTE is non-zero, but the Present bit is 0. The kernel deliberately sets up this odd situation to indicate an anonymous page which has been `swapped out`. The page frame number does not correspond to a real address, but is used as an identifier to help the swap subsystem find the image of the page in the swap area. We will need to initiate an I/O operation to swap-in the page image to a new page frame. This will be discussed further under "Swap Space".
• The Present bit is 1. At this point in the code, we know that it must be a hardware protection violation which tripped the page fault, and we have previously checked that the attempted access is valid for this region. Therefore we need to resolve the copy-on-write

as discussed above.



Note that in the above diagram, the order of the bits within the PFN has been abstracted to make a clearer presention. If you dare to look at the X86 documentation, you'd find it differs.

Page Frame Reclamation / Paging-Out

We have seen that the kernel has nearly constant demands for free page frames. The Page Frame Pool is the set of page frames that are free for allocation. The kernel never wants to let that pool run dry, and actually wants to make sure that there is always something in the pool for a little insurance. Getting caught with an empty pool at the time of need is a very awkward situation for the kernel, one which, as we've seen, causes processes to die badly with `SIGKILL`s.

When physical RAM is plentiful, then all of the pages that have been demand-paged in by all of the processes (that are still running and have not exec'd) would still be in memory. We call the total number of page frames mapped to a given process the **Resident Set Size (RSS)**. As the process continues to run and demand-page more of its address space, the RSS will increase, and the pool will decrease, but with plentiful RAM, the pool will never get critically low. In time, processes will either exit or exec, and their Resident Sets would get freed, replenishing the pool.

However, resource scarcity is what creates interesting problems in Computer Science. When physical RAM is constrained, there would come a point where the total RSS's of all the processes has the pool almost dry, which greatly alarms the kernel. Therefore, the kernel pro-actively manages the page frame pool to avoid this situation, by "stealing" pages that were already mapped in to processes and returning them to the free pool. This is known as the **Page Frame Reclamation Algorithm** (PFRA).

We can think of the pool of page frames as a cache for actively-used data which are otherwise stored on disk (in a file or in swap space). So, the PFRA problem is congruent to the cache eviction problem in Computer Architecture. Ideally, we'd like to achieve **Least-Recently Used** (LRU) behavior: determine which pages have been accessed when, and reclaim the pages that have been idle the longest. The reason for this is that things which have been recently used will probably be used again soon, and things which have sat idle for a long time will probably be idle for quite a bit longer. The actual Linux PFRA only approximates this behavior, because to achieve strict LRU, we'd need the hardware to keep some kind of clock tick field in the PTE and update it with each access. Rather, we've seen that the hardware only maintains a boolean ACCESSED bit, so we'll have to improvise.

The PFRA is implemented within the Linux kernel using what is known as a **kernel task**. A kernel task (aka kernel thread) is like a user-mode process, but it never leaves kernel mode. It can make system calls, using a slightly modified, internalized syntax. Kernel tasks are scheduled by the scheduler, can go to sleep and wakeup, etc. They do not have their own address space but that is OK since they only access the shared kernel memory area. The name of this task is `kswapd` and it can be seen as a process, of sorts, via the output of the `ps` command.

When RAM is plentiful, the PFRA/kswapd is sleeping. There are two parameters which can be tuned by the system administrator which are known as the low-water and high-water marks for the page frame pool. When pool dips below the low-water mark, this awakens kswapd, and it starts looking around for pages to steal. It continues running until it has freed up enough memory that the pool is above the high-water mark, and then goes back to sleep.

The Page Frame Reclamation Algorithm (PFRA) maintains two lists of non-free page frames: active and inactive. These lists are chained via the `lru` field of the page descriptor. The PFRA scans the page descriptors on the active list with the desire to find hypoactive pages and move them to the inactive list. It uses the reverse mapping (see below) in the page descriptor to find all the PTEs mapping to this physical page (when the page is shared there will be multiple PTE pointing to the same physical page). Each PTE is examined to see if the ACCESSED bit is set, then the ACCESSED bit is cleared. The PFRA also keeps a second bitwise flag, `PG_referenced`, in the `flags` field of the page descriptor, which stores the previous value of the ACCESSED PTE flag. (when there are multiple PTEs mapping the same page, PG_referenced is the OR of the ACCESSED PTE bits).

When an page which had `PG_referenced` set to 0 is visited again and the ACCESSED PTE flag(s) is (are all) still clear, it means the page has not been used in two successive scan passes. The PFRA will then move the page to the tail of the *inactive list*.

After the PFRA finishes hunting for active pages to move to the inactive list, it starts trying to pull victims off the inactive list, starting at the head of the list. I.e. it starts with the pages which have been sitting on the inactive list the longest. Thus the PFRA approximates LRU ordering.

For each inactive page scanned, the ACCESSED bit(s) is (are) checked again, one last time. Note that this bit is only cleared when the PFRA has visited the page from the *active* list, therefore if there has been any access whatsoever between the time that the page was moved to inactive and the time it was visited on the inactive list, this PTE bit will be set, and the page is just moved back to the active list. Otherwise, the page will be reclaimed. If the PTE indicates that the page is dirty, a write-back to backing store is begun. This operation might take a while, so `kswapd` continues to look at additional pages that are not dirty, until it has reclaimed the desired number of pages. Pages which are not dirty can be reclaimed by simply immediately unmapping the PTE(s). Pages which were in the process of being written back are unmapped when that writeback operation concludes.

Pages that the PFRA has reclaimed are then placed on the free list. Since they are placed at the tail of the free list (for a given buddy order) and the page allocator pulls from the front, this further enhances the notion of Least Recently Used. Pages which are on the free list and haven’t been re-used but still contain valid images can still be pulled back off so that a page fault can be resolved as a minor fault.

The PFRA is adaptive and if it feels that it isn’t reclaiming enough memory, it increases the rate at which it scans active pages, thus making the activity timeout window shorter and increasing the pool of potential inactive pages. This is a delicate balancing act because the PFRA/`kswapd` consumes CPU resources in doing all of this scanning. Excessive scanning wastes CPU time, but insufficient reclamation can cause memory starvation and severe performance penalties.

Like scheduling, memory allocation and reclamation algorithms are the subject of extensive research and development.

Reverse Mapping

For any given page frame, there may be multiple mappings for it. The kernel must keep track of all of them and prevent inconsistent views. For example, a given page may appear as part of the virtual address space of one or more processes, correspond to a particular offset in a particular memory-mapped file, and furthermore correspond to a particular offset within the hard disk storing that file. The PFRA, when visiting page frames, must be able to find all existing mappings (PTEs) of that page so that it can locate and possibly modify the PTEs. To visit the PTEs, the PFRA must determine the `vm_area_struct` and offset within that region (there could be multiple regions).

Either a page is mapped to a file, or it is anonymous. These two conditions are mutually exclusive. The `mapping` field of the page descriptor contains a pointer to one of two objects which will help with the reverse mapping. If the least significant bit is 1, the page is anonymous and `mapping &~01` points to a `struct anon_vma`. If the lsb is 0, the page maps to a file and `mapping` points to a `struct address_space` object for the file. This use of the least significant bit is a programming trick: because kernel pointers are always aligned on 4-byte boundaries, the least significant bit can be wasted for this purpose.

Anonymous Mappings

The `anon_vma` is simply the head of a doubly-linked circular list of memory region descriptors (`vm_area_struct`), chaining through the `anon_vma_node` pointers in the latter. Furthermore, the `vm_area_struct` contains a pointer (`anon_vma`) back to the `anon_vma` list head. More often than not, the list contains just one memory region descriptor. However, when a region becomes shared (e.g. after a fork) this list contains each region in each process which refers to it. This will also be the case for MAP_SHARED regions that were inherited via fork. Therefore, each `anon_vma` structure represents one anonymous virtual memory region (which might be shared among multiple processes).

A process data region or other MAP_PRIVATE file mappings might find itself on the `anon_vma` list and also part of a file mapping, because some pages in the region still correspond to the file, but some have been written to making them anonymous. Within this region, mapped-in pages are either "virgin" and still correspond to the original file, or they have been dirtied and are now backed by swap space. This distinction is captured via the least significant bit of the `mapping` field of each page descriptor as described above.

The PFRA, after getting the `anon_vma` list start from the page descriptor, can walk the linked list anchored by the `struct anon_vma` and visit each memory region (`vm_area_struct`) which maps that page, which gives the starting virtual address of the that region. The `index` field of the page descriptor gives the offset, in pages, of that particular page within the region. The `pgd` field of the `struct mm_struct`, which in turn is accessible from the vm_area_struct, gives the address of the page global directory for the process. It is now a trivial matter to walk through the page table structure and find the PTE.

Swap Space and anonymous page-outs

When a dirty anonymous page is selected for reclamation, it must be saved someplace before reclaiming the page. Since we can't save it to a file, it must be saved to **swap space**. The kernel maintains one or more swap areas, which are either raw disk partitions, or files, in either case prepared with the system utility command `mkswap`, and then made available as part of swap space with the `swapon` command.

Anonymous pages aren't meant to be persistent, and so swap space is just a pool of disk space where we can stash them. When the system reboots, that part of disk is not preserved. There is nothing within a swap area which would tell one the identities of the page images. That information is kept by the kernel in RAM and simply discarded when the system halts. The kernel maintains an array of swap area descriptors, each of which includes a free map. Each slot of a free map gives the reference count of the corresponding page-sized slot of the corresponding swap area.

A particular page frame containing an anonymous page may be mapped by multiple PTEs in one or more processes. Therefore, when that page frame is swapped out, ALL of the PTE must be marked as not present. A free slot in some swap area is chosen (the algorithm to do so keeps things such as disk I/O performance in mind). Free slots are indicated by a swap map entry of 0. Once the slot is allocated, the swap map entry will reflect the number of mappings that had been pointing at that shared page frame.

Now the Linux kernel uses a slightly dirty trick. The swap area number (a 7-bit integer) combined with the slot number within the area form a unique identifier of the swap slot. 24 bits are allowed for the slot number, meaning the maximum size of one swap area is 2ˆ24*2ˆ12=2ˆ36 or 64 GB. The total amount of swap space on the system is limited to 2ˆ(36+7) = 8 TB.

Given this information, the page fault handler can, when the page is once again referenced, find the corresponding swap slot on disk and read it in. This swap identifier is stored in each PTE which had mapped the swapped-out page. There are 31 bits in the identifier and a PTE is 32 bits. The Linux kernel packs the bits into the PTE in such a way that the Present flag is always 0, and the remaining 31 bits are scattered between the PFN and the Flags fields. It isn't a valid PTE but since the Present bit is OFF, the hardware will never look at it or touch it. (On X86-64, there are 63 bits of the PTE to play with for encoding a swap area and swap slot....that's a LOT of swap space!)

A page that was previously swapped out and is now faulted back in from swap space gets flagged as DIRTY by the kernel, even if this page fault is just for reading. This is because the underlying virtual page image is in fact dirty. By virtue of the fact that it was in swap space, it is no longer the virgin (all-0) anonymous page. It could be the case that this page will get reclaimed again, but unlike file-mapped pages, where there is a definite spot in the file corresponding to the page, anonymous pages don't "own" a particular swap slot. So, we'd have to write the page out, again, to a new swap slot.

File Mappings

For files which are mapped into memory, it is important to be able to locate quickly the page frame which currently contains a particular (page-aligned) offset in the file, and/or to determine that no such page is currently resident. Consider the `read` system call, the semantics of which must work correctly and transparently with memory-mapped files. `read` amounts to finding the page frame which holds that part of the file (if there is no such frame, ask the filesystem to bring it in from disk and sleep until that happens), and then copying the requested portion of that page into the user buffer.

Likewise, when faulting-in a memory-mapped file, it could be that the same (page-aligned) area of the file has recently been accessed from the same or different process using the read or write system call, or via an mmap'ing, in which case the fault can be resolved a minor fault by pointing the PTE at the existing page frame which contains that image.

An `address_space` structure is used to provide this quick lookup. One of these is allocated for every open inode. It has many fields, but among them are two types of trees. One is a flexible radix-64 tree which, given an offset (in page-sized units) into the file, finds the struct page corresponding to that part of the file, or NULL if there is none such. Each node of the radix-64 tree has 64 pointers. If the file size is 256K or less (64*4096) then the entire file's address space is handled by a single-level tree. Otherwise, the tree grows. A two-level tree can handle (64*64*4096) or 16MB of file addressing, etc. up to 6 levels.

```
struct address_space {                /* /usr/src/linux/include/linux/fs.h */
        struct inode            *host;          /* owner: inode, block_device */
        struct radix_tree_root  page_tree;      /* radix tree of all pages */
        spinlock_t              tree_lock;      /* and lock protecting it */
        unsigned int            i_mmap_writable;/* count VM_SHARED mappings */
        struct prio_tree_root   i_mmap;         /* tree of private and shared mappings */
        spinlock_t              i_mmap_lock;    /* protect tree, count, list */
        unsigned long           nrpages;        /* number of total pages */
          /* some other fields elided for simplicity */
};
```

PST & Reverse Mapping for file-mapped regions

While the radix tree is great for going from a known offset in the file to a page descriptor, the PFRA is trying to go a different way: find all PTEs which refer to this particular offset within the file. The `address_space` pointer of the `struct page` identifies the file, and the `index` field gives the offset in pages within the file. A simple approach would be to link all regions mapping to the file into a linked list, as is done for the `anon_vma` structure. However, while sharing of an anonymous region is usually limited to a small number of processes, memory-mapped files, especially those containing a common executable, are shared by many processes. There can also potentially be multiple mappings to different parts of one file (consider an a.out file where the .text and .data section are mapped separately).

Linux uses a Priority Search Tree for *each file*. It is accessed via the `address_space` structure's `i_mmap` pointer, and can quickly answer the following question: "given a particular offset within the file, tell me all memory regions which contain it." I.e. the PST sorts the mapping regions by their starting offset. The PFRA examines all memory region descriptors (`vm_area_struct`) which map to the page. Each descriptor contains the starting virtual address of the region, and the corresponding offset within the file (this is in the `vm_pgoff` field). Now the offset of the page within the region can be readily calculated, and from there the PTE can be found.

Paging-out file-mapped page frames

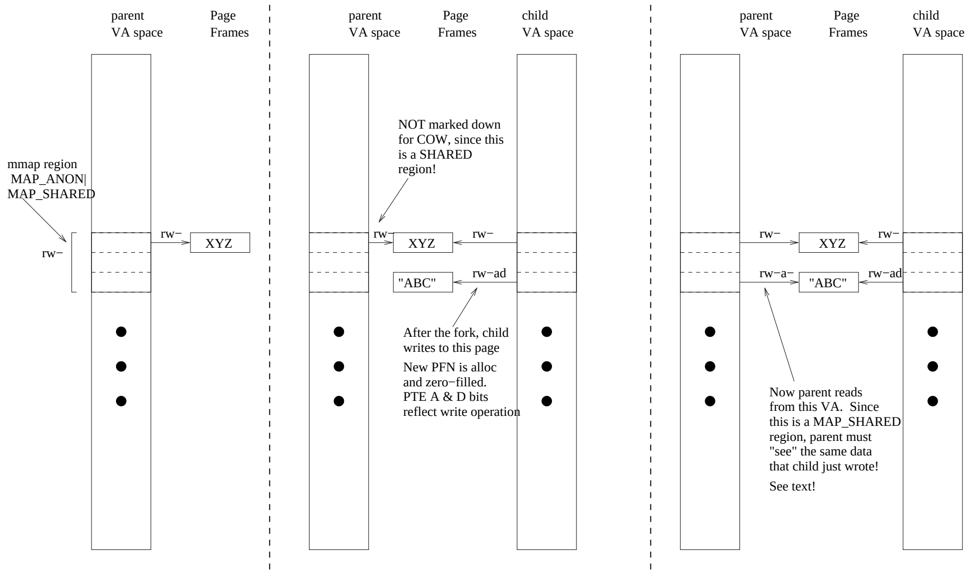
A file-mapped page which needs to be paged out and which is dirty must be part of a MAP_SHARED file-mapped region (if it were MAP_PRIVATE, the dirty pages would be paged-out to swap). We know from the `mapping` and the `index` fields of the `struct page` the file and the offset within that file where the page needs to go. So paging-out a file-mapped dirty page involves asking the underlying filesystem module to write the page to that offset within the file on the disk.

A special case: MAP_ANONYMOUS|MAP_SHARED

When an mmap region is established with MAP_ANONYMOUS | MAP_SHARED, it is clearly anonymous: no part of any file corresponds to it. But in order for it to work correctly when shared among multiple address spaces, the Linux kernel must resort to a trick. Consider what would happen if such a region is created and then is being shared by two processes after a fork. Process A demand-pages in a page of this region for the first time. Since it is anonymous, the kernel satisfies this as a minor fault with a zero-filled page frame. Let us further suppose that A writes to this page. At some later time, process B also tries to access, for the first time, the same virtual memory address that A did. In order for MAP_SHARED to work correctly, B must **not** get a zero-filled page frame! Instead it must get the same page frame that A has.

To implement this, the Linux kernel treats this region as if it were mapped to a file, but not a file that is on disk or has a pathname in the ordinary sense. The kernel creates a one-time inode that is associated with the `/dev/zero` special file, which is a character device that appears to supply endless 0 bytes. The inode is disconnected from pathname space so there is no way another process could ever access it. Mapping to an instance of `/dev/zero` satisifies the idea that new pages of this MAP_ANON|MAP_SHARED region should be 0-filled, but also preserves the sense that each such mapping is distinct. Now in the above example, when B incurs its page fault, this will be treated by the kernel as a forward-mapping file-mapped case. The kernel will consult this isolated inode and look at its `address_space` structure, from which the page frame that already contains the correct image of the region (currently mapped in A) will be found. It is then a simple minor page fault to hook up this PFN to B.

However, `/dev/zero` is not a real file. When page frames are associated with it, they are marked as anonymous, despite their file-mapped apparent origin. If a page frame associated with this MAP_ANON|MAP_SHARED region gets reclaimed, this will get written back to *swap space*, just like an anonymous page. Some special magic is then needed when/if the page is faulted back in to make sure the `address_space` correctly reflects its existence.


parent VA space
Page Frames
mmap region MAP_ANON| MAP_SHARED
rw−
rw−
XYZ
parent VA space
Page Frames
child VA space
NOT marked down for COW, since this is a SHARED region!
rw
XYZ
rw−
"ABC"
rw−ad
After the fork, child writes to this page
New PFN is alloc and zero−filled. PTE A & D bits reflect write operation
parent VA space
Page Frames
child VA space
rw−
XYZ
rw−
rw−a−
"ABC"
rw−ad
Now parent reads from this VA. Since this is a MAP_SHARED region, parent must "see" the same data that child just wrote!
See text!

Resource Exhaustion

In an ideal world, systems would always have sufficient resources to handle the jobs which are thrown at them. But that world is not a terribly interesting one in terms of operating systems design. In reality, memory resource exhaustion is often a problem. We can consider two distinct problem areas: Insufficient physical memory, and insufficient swap space.

Thrashing

When memory becomes very scarce, the system can spend much of its time freeing up pages by swapping-out, followed by swapping-in when a process which has had its pages stolen gets scheduled and tries to run. This is an age-old problem known as **thrashing**. Older versions of Linux tried to ameliorate it by using what Linux called a "swap token". A process which holds the swap token is given a temporary reprieve by the PFRA. The swap token is given to a process based on heuristic rules...basically a process which has recently been victimized by the PFRA, has good priority and looks like it will do something useful if allowed to run.

This isn’t necessarily a perfect solution. To contrast, the Solaris kernel takes a different approach. When memory runs low, a kernel thread called the "swapper" picks entire processes to victimize and steals all of the process’ pages. A process thus swapped-out is basically put to sleep and therefore doesn’t get scheduled. When things get calmer, the process is released from the swapper’s grasp and is allowed to run and fault its pages back in.

More recent versions of Linux use a sophisticated framework known as Control Groups (cgroup) to provide memory management policies. This allows groups of processes to be considered together in terms of their memory usage, and to determine which groups have priority. When memory is low, low-priority cgroups would be targeted first for page reclaims.

Memory over-commit and the out-of-memory killer

Just like an airline which over-books seats, it is possible for the kernel to hand out more virtual memory than it has physical memory and swap to cover. If it happens that all of the memory is demanded at once and there are insufficient free swap slots to page-out to, the kernel will have no choice but to kill processes in order to regain memory. There is an Out-Of-Memory killer which is triggered when the system becomes desperately low of memory. It picks large processes which haven’t been running too long.

This might seem like a design flaw. After all, the programmer successfully allocated memory using malloc (which in turn used `brk` or `mmap`), so why has the kernel gone back on its promise and why should a process be terminated because of something which is not its fault?

On the other hand, the most conservative approach, which guarantees that one will never run out of memory, is to reserve a swap space slot for each virtual anonymous page allocated. Then there will always be a place to page-out any resident page and we can never get into this trouble. The problem with this approach is that processes will start getting memory allocation failures long before the system has reached a high probability of running out of memory, unless the administrator has configured a very large amount of swap space. The reason: most virtual anonymous pages are never actually paged-out. Consider a large process which forks and then execs. Most of the virtual address space of the child will be discarded.

Older Linux kernels took the first approach. Solaris kernels tend towards to more conservative approach. Modern (2.6+) Linux kernels allow the administrator to tune the memory over-commit policy.

```
The following is verbatim from \fCDocumentation/vm/overcommit-accounting\fP:

The Linux kernel supports the following overcommit handling modes

0       -       Heuristic overcommit handling. Obvious overcommits of
                address space are refused. Used for a typical system. It
                ensures a seriously wild allocation fails while allowing
                overcommit to reduce swap usage.  root is allowed to
                allocate slighly more memory in this mode. This is the
                default.

1       -       Always overcommit. Appropriate for some scientific
                applications.

2       -       Don’t overcommit. The total address space commit
                for the system is not permitted to exceed swap + a
                configurable percentage (default is 50) of physical RAM.
                Depending on the percentage you use, in most situations
                this means a process will not be killed while accessing
                pages but will receive errors on memory allocation as
                appropriate.

The overcommit policy is set via the sysctl vercommit_memory’.

The overcommit percentage is set via vercommit_ratio’.

The current overcommit limit and amount committed are viewable in
/proc/meminfo as CommitLimit and Committed_AS respectively.

Gotchas
-------

The C language stack growth does an implicit mremap. If you want absolute
guarantees and run close to the edge you MUST mmap your stack for the
largest size you think you will need. For typical stack usage this does
not matter much but it’s a corner case if you really really care

In mode 2 the MAP_NORESERVE flag is ignored.


How It Works
------------

The overcommit is based on the following rules
```

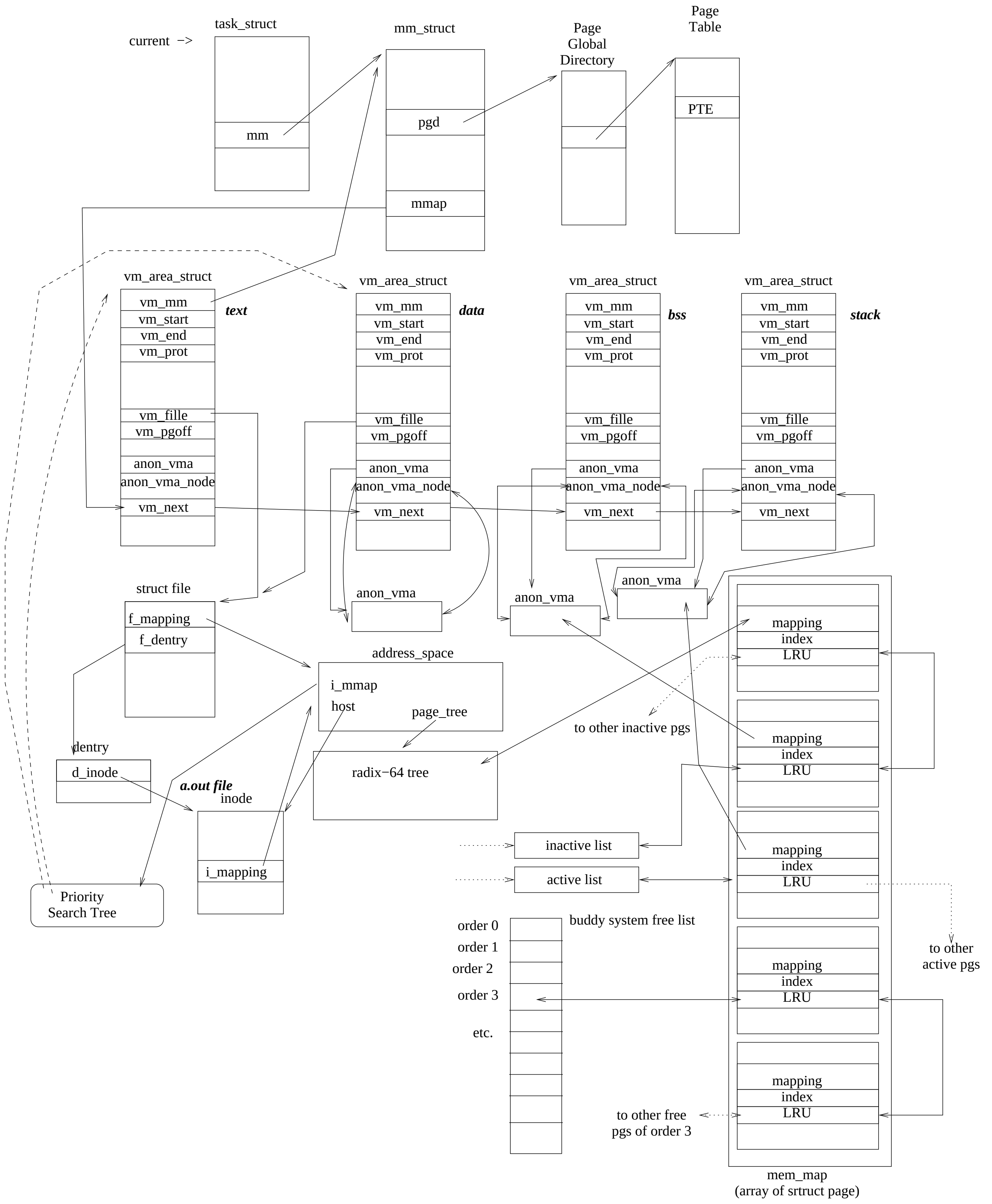
```
For a file backed map
        SHARED or READ-only     -       0 cost (the file is the map not swap)
        PRIVATE WRITABLE        -       size of mapping per instance

For an anonymous or /dev/zero map
        SHARED                  -       size of mapping
        PRIVATE READ-only       -       0 cost (but of little use)
        PRIVATE WRITABLE        -       size of mapping per instance

Additional accounting
        Pages made writable copies by mmap
        shmfs memory drawn from the same pool
```

The /proc interface

Under the Linux kernel, we can get a lot of internal information about the memory subsystem via the `/proc` interface. Here are some of the fun things:
• `/proc/pid/maps` : a human-readable listing of all the virtual memory regions (vm_area_struct). Each line represents one region and contains: the start and end virtual address, some of the VM_flags (READ,WRITE,EXEC and SHARED vs PRIVATE), and if the region is file-backed, the offset, device and inode number as well as the pathname of the file.
• `/proc/pid/smaps` : same info as above, but in an expanded format which shows all of the VM_XXX flags, and a count of how many PTEs are mapped in, and how many are currently dirty, plus a bunch more interesting stuff. `/proc/pid/numa_maps`: similar to `maps`, one line per region, summarizes number of PTE mappings and how many are dirty. Has some other stuff regarding NUMA (non-uniform memory architecture) that is way beyond this course!
• `/proc/pid/pagemap`: This is a binary file consisting of a series of 64-bit entries, each of which provides information about the corresponding PTE in the virtual address space. Not all of the PTE is exposed unfortunately, but we can see the Page Frame Number part of the PTE and a few other properties.
• `/proc/kpageflags`: This binary file is global (not per-PID) because it is indexed by PFN. It contains a series of 64-bit entries, in PFN order, which encode the PG_XXX page frame flags.
• `/proc/kpagecount`: A similar file which contains the `mapcount` field of the struct page.

A summary of data structures

On the next page is a very complicated diagram showing the relationships between the various kernel data structures that keep track of virtual memory. It does not represent every possible situation, but shows us one process with the usual 4 memory regions.


current ->
task_struct
mm
mm_struct
pgd
mmap
Page Global Directory
Page Table
PTE
vm_area_struct
vm_mm
vm_start
vm_end
vm_prot
vm_fille
vm_pgoff
anon_vma
anon_vma_node
vm_next
text
data
bss
stack
struct file
f_mapping
f_dentry
dentry
d_inode
a.out file
inode
i_mapping
Priority Search Tree
address_space
i_mmap
host
page_tree
radix−64 tree
anon_vma
to other inactive pgs
inactive list
active list
buddy system free list
order 0
order 1
order 2
order 3
etc.
mapping
index
LRU
to other active pgs
to other free pgs of order 3
mem_map (array of srtruct page)

Appendix I: Spectre, Meltdown, and Page Tables

In 2018, a series of security vulnerabilities were found in X86 processors that have colloquially become known as "Spectre / Meltdown." The underlying Computer Architecture concepts are fairly advanced and beyond the scope of this course. However, in summary, these vulnerabilities all involve speculative execution, where instructions (or portions thereof) are fetched, and/or operands are fetched, in advance of the actual instruction firing off, to improve pipeline performance. The essence of the security bug is that proper checking of the User/Supervisor privilege bit(s) of a PTE is not done during this speculative memory access. As a result, data which are in kernel memory could potentially be read out by user-mode processes.

This is a pervasive problem and unfortunately all solutions to it involve some reduction in performance. The solutions involve both microcode changes to the processor hardware and changes to the kernel. Not only Linux, but other UNIX variants, and Windows, are all vulnerable to this and have all had to release emergency security patches.

From a kernel standpoint, the most significant change has been the elimination of a shared user/kernel page table. This strategy, in which the entirety of the kernel’s virtual address space is visible in the page table structure, but locked out by setting the PTEs to be Privileged, has been replaced by one in which only a small amount of kernel virtual address space overlaps into the page tables which user-mode processes see. This small VA space is necessary for interrupt/fault/system call processing. Upon entry to the kernel, the %cr3 register is switched to the kernel’s address space (again, a single shared address space for the entire kernel). On CPUs with the Address Space ID / Process Context ID feature, this is also updated, just as with any address space change, and the performance penalty is not that severe. However, lacking that feature, older hardware suffers greatly, as every re-entry into the kernel is accompanied by a rash of TLB misses.

Appendix II: How an executable file is created

In the original UNIX kernel, each program was entirely **statically-linked**, i.e. all of the code required to execute the program, and in particular all of the libraries which the program requires, are put together at compile-time into one large monolithic executable file. This practice began to change during the 1990s, and today almost all programs in almost all UNIX-derived systems are *dynamically linked*. It is still possible to compile a program which is statically linked, but this is exceptional. We will first discuss how statically linked programs are linked together, and then delve into dynamic linking.

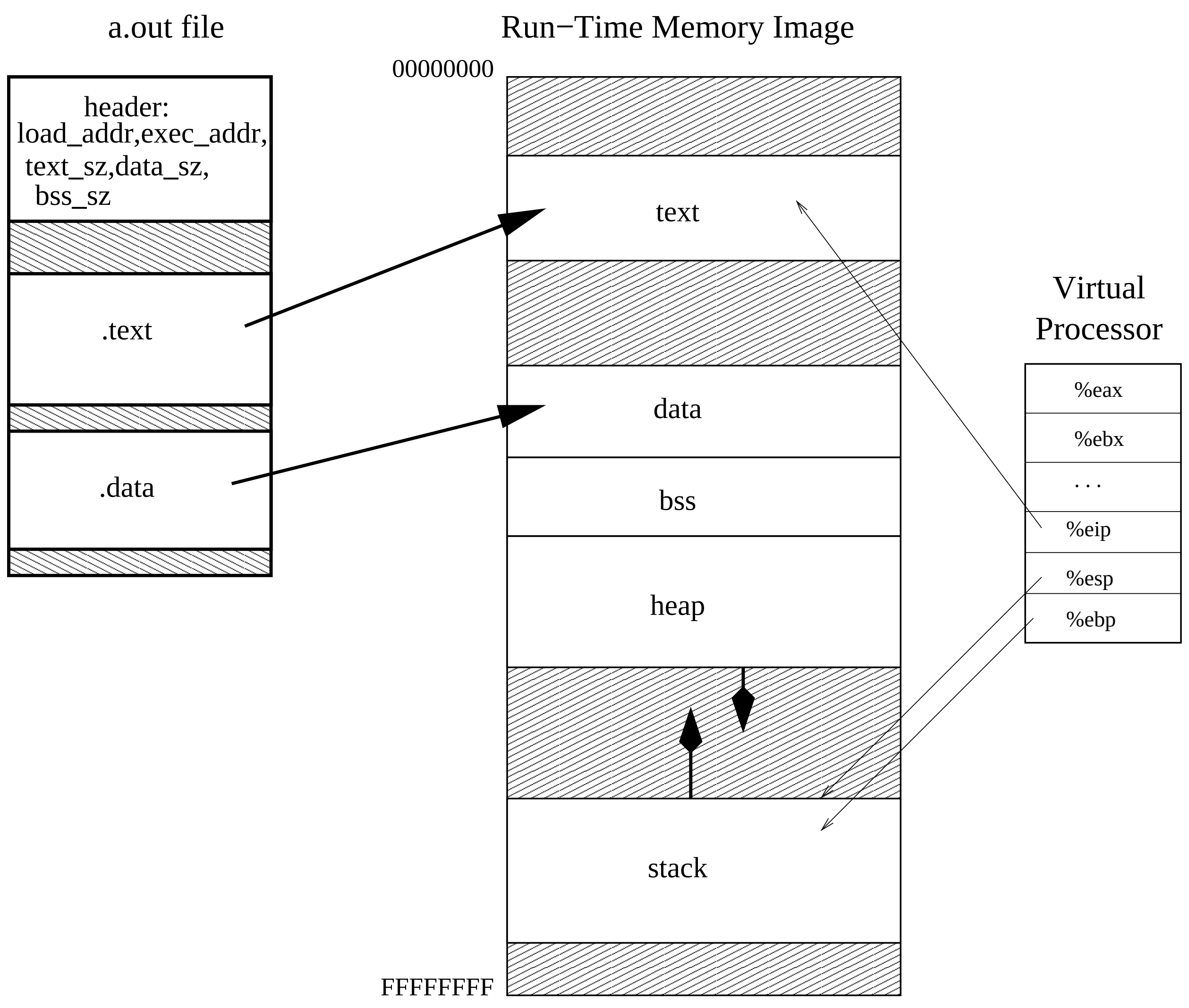
Compiling a C program into an executable file is a multi-stage process involving several tools. When one simply runs:

```
cc test.c
```

The `cc` command transparently executes the tools, detailed below, resulting in an executable file called `a.out` (if there are no fatal errors in compilation). The "a" in "a.out" stands for "Absolute", meaning that all symbolic references have been resolved. An a.out file contains pure machine code that can be directly executed by the processor. The executable file ("a.out") contains everything that the operating system needs to create a program's initial memory configuration and begin execution. The header of the a.out identifies it to the operating system as an executable file and specifies the processor architecture on which the machine code instructions will run. A program compiled for an x86 architecture can not be executed, for example, on a SPARC processor. The header also gives the size of the text, data and bss memory requirements, as explained below.

The `a.out` file includes the literal bytes of executable code which will be loaded into the process's `text` region. These are in a contiguous part of the `a.out` file. The header gives the offset of the beginning of the `.text` section of the `a.out`, and also the **load address**, i.e. the virtual address at which the compilation system expects this text to be loaded.

Likewise, there is a `.data` section in the `a.out` which contains an image of what the `data` region of the process will look like at program startup.


a.out file
header:
load_addr,exec_addr,
text_sz,data_sz,
bss_sz
.text
.data
Run−Time Memory Image
00000000
text
data
bss
heap
stack
FFFFFFFF
Virtual
Processor
%eax
%ebx
. . .
%eip
%esp
%ebp

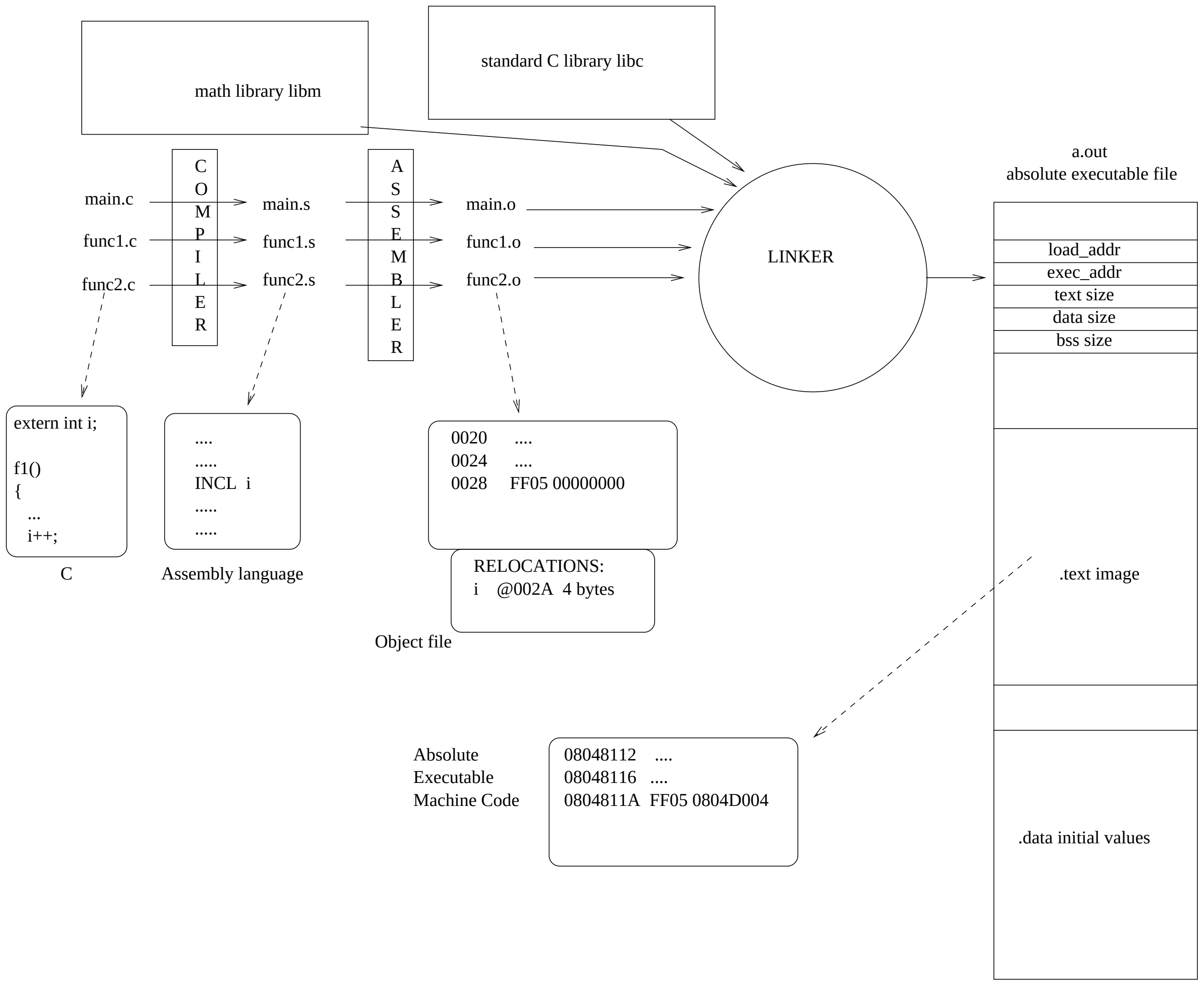
The **bss** section contains all un-initialized global variables. The C language specification states that all such variables, lacking an explicit initializer, must be initialized by the operating system or C run-time environment to 0.

```
int j=2;
int k;
main()
{
 int l;
         /*...*/
}
```

In the example above, `j` has an explicit initializer, and will be in the data segment. The value of the initializer will be found in the .data section of the a.out. `k` is uninitialized. Therefore it will reside in the bss segment of memory and will have an initial value of 0. The ISO C standard says that "If an object that has static storage duration is not initialized explicitly, it is initialized impliclitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a NULL pointer constant". This is satisfied by filling every byte of the bss region with 0, and is performed by the operating system. The variable `l` has automatic storage scope, and therefore will be found on the stack (or, if the compiler is set for heavy optimization and a pointer to `l` is never taken, it may live in a register). Automatic variables are not 0-initialized.

Behind cc-losed doors

The `cc` command is often and erroneously called "the compiler." In fact, it is a wrapper program which invokes a series of tools. Each tool transforms a specific kind of input file to a specific kind of output file. It is possible to instruct `cc` to interrupt the process at any stage, and to access and manipulate these intermediate files as needed.



The compiler proper takes C code, passing it through the macro pre-processor, and compiles it into symbolic assembly language. By using the `-S` option to `cc`, we can halt the process before the assembler and linker are invoked, and view the intermediate assembly language file, which will have a `.s` extension.

The assembler takes symbolic assembly language files (`.s`) and converts them into relocatable object files (which have a `.o` extension). An object file is similar to an executable ("a.out") file in that it contains machine code, however symbols which can not be resolved (as detailed below) are left dangling. Therefore, it is not ready to run until these dangling links are settled, possibly by linking with additional object files. By giving `cc` the `-c` option, the linker will not be run and the `.o` files will be left for subsequent examination or linking. It is also possible to invoke the assembler directly with the `as` command.

Although some compilers do not separate the assembler, it is useful to divorce it from the compiler and make it a separate utility. In this way, a common assembler program can be written and employed by compilers for different languages. For example, the GNU compilers for C, C++ and FORTRAN are all different programs, yet they all call on the `as` program for assembly. It is also useful to be able to introduce `.s` files that have been written by hand for certain performance optimizations or platform-specific features.

The linker takes multiple object (`.o`) files (or libraries of object files, with `.a` extensions, as discussed below), resolves symbol references, and creates an absolute executable file. If unresolved symbols remain at this point, the linker will not be able to create an absolute executable and an error will result.

Like the assembler, the linker is a separate tool which can be called directly as `ld`. The reason for this rather un-mnemonic two-letter name is that historically, the linker has also been called the "loader", because it was responsible for creating the "load deck" of punch cards that could then be directly run by the computer. Besides, `ln` was already taken as a command name.

When compiling C programs, it is best to allow the `cc` wrapper to invoke `ld` rather than trying to do it manually as there are a number of complex platform-specific options needed to actually create a run-able program.

The `.o` files need not have come from C source code. It is possible to distribute binary object files or libraries which can be linked against user-supplied code. It is also possible to create cross-language executables, where part of the code is written directly in low-level assembly language, or in another high-level language (with all due caution about mixed run-time environment expectations).

Separate Compilation

Very simple C programs can be contained in a single `.c` file. As the size of the project increases, it becomes extremely inefficient to do this. The tasks of pre-processing, compiling and assembling are CPU-intensive. With a single file, any change, no matter how small, requires that the entire file be recompiled.

The program can be split up into a number of smaller `.c` files. There are varying opinions on how finely to divide the program and on what basis to select which parts go into which file. Some guidelines:

1) Groups of functions which form a coherent subsystem are good candidates for including in their own file. In C++ terms, this would be a class. Some subsystems are so complex that they need to be further subdivided.

2) Whenever a single `.c` file is on the order of 1000 lines of code, it is probably getting

too big. On the other hand, having lots of files each containing one or two 10-line functions is also wasteful.

3) Code which is being incorporated from another source, or which is being worked on by a different programmer, should have its own file, or perhaps be structured as a library, to make it easier to deal with changes.

As an example, let's say our program has been divided into files `f1.c`, `f2.c` and `f3.c`. To compile this, we could run: `cc f1.c f2.c f3.c`, which would perform the entire pre-processing, compiling and assembling process on each of the three files, then invoke the linker to create the `a.out`. However, suppose we make a change to `f1.c` only. Why do all that work again for the other two files, which have not changed?

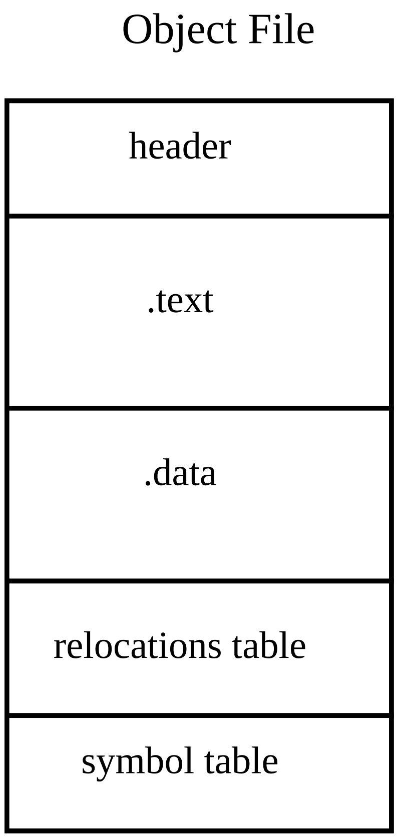
Instead, we can first run: `cc -c f1.c f2.c f3.c`, which will pre-process, compile and assemble each of the files, creating `f1.o`, `f2.o` and `f3.o`. Then we run: `cc f1.o f2.o f3.o`. The `cc` wrapper sees that we are giving it `.o` files and passes them directly to the linker. Now, if we change `f1.c`, we run `cc -c f1.c` to re-compile it alone, and then run `cc f1.o f2.o f3.o` again to link the program. This process is automated by the tool `make`, which uses dependency rules and file modification times to determine which compilation phases to invoke and on which files.

The linker vs the compiler

The job of the linker is to take one or more relocatable object files, resolve symbol references, and create an absolute executable file. The linker does not know about C's type algebra, nor should it, since the linker is independent of any particular high-level language.

Unfortunately for the student, the C language also deals in symbols, e.g. variable and function names, and this issue is sometimes difficult to separate from the symbol processing which happens at link time. Here are some things that never appear in an object file and do not concern the linker:
• C language type specifiers (such as "p is a pointer to a pointer to a function returning int and taking two int arguments")
• C language structure, union and enum definitions and typedef names.
• `goto` labels
• Internal labels that may be generated by the compiler for loops, switch statements, if statements, etc.
• automatic (local) variables. An important exception is a local variable that is declared with the `static` storage class. This variable has local scope, i.e. its name is not visible outside of the curly-braced block in which it is declared, but it lives in the same

neighborhood as global variables.



What’s in a .o?

Let’s see what is actually in an object file and how it gets there:

```
/* f1.c */
extern int i;

f()
{
         i=2;
}

/* f2.c */
int i;
main()
{
         i=1;
         f();
}
```

In `f1.c`, the `extern` storage class for variable *i* tells the compiler that this variable is external to that `.c` file. Therefore, the compiler does not complain when it does not see a declaration that variable. Instead, it knows that `i` is a global variable, and should be accessed by using an absolute addressing mode. Neither the compiler nor the assembler knows what that actual address will be. That is not decided until the linker puts all the object files together and assigns addresses to symbols. Therefore, the assembler must leave a "place-holder" in the object file. Likewise, in f2.o, there will be a reference to the symbol `f`.

Let us examine the assembly language files produced (using an older gcc under Linux on an x86 system):

```
 **** f1.s
         .text
         .globl f
        .type    f,@function
 f:
        pushl %ebp          *These instructions set up
        movl %esp,%ebp                *the stack frame pointer
        movl $2,i
        leave                         *Restore frame pointer
        ret
 .Lfe1:
        .size    f,.Lfe1-f

 **** f2.s
         .text
         .globl main
        .type    main,@function
 main:
        pushl %ebp          *These instructions set up
        movl %esp,%ebp                *the stack frame pointer
         movl $1,i
        call f
        leave                         *Restore frame pointer
        ret
 .Lfe1:
        .size    main,.Lfe1-main
        .comm   i,4,4
```

The assembler directive `.text` tells the assembler that it is assembling opcodes to go in the .text section of the object file. The `.globl` directive will cause the assembler to export the associated symbol as a defining instance. `.type` is used to pass along information into the object file as to the type of the symbol. Please note that it has nothing to do with the C language notion of type. Symbol types may either be functions or variables. The linker is able to catch gross errors such as if `f` were defined as a variable in `f1.c` instead of a function. The `.size` directive calculates the size of the function by subtracting the value of the symbol representing the first instruction of the function (e.g. `main`) from a placeholder assembler symbol (e.g. `.Lfe1`) representing the end of the function. Note that the CALL to function `f` is done symbolically, as is the assignment into global variable `i`.

The resulting object file is similar to an a.out file, however all addresses are relative. In addition, the object file will have a section known as the **symbol table** containing an entry for every symbol that is either defined or referenced in the file, and another section called the **relocations table**, described below.

Now, let us view the .text section of `f1.o` (the listing below was re-formatted from the output of objdump -d):

```
Offset Opcodes                              Disassembly
0000   55                           pushl %ebp
0001   89E5                         movl %esp,%ebp
0003   C7050000000002000000         movl $2,0
000D   C9                           leave
000E   C3                           ret
```

First, note that the offset of the first instruction is 0. Obviously, this can not be a valid memory address. All offsets in object files are relative to the object file. It isn't until the linker kicks in that symbols gain absolute, usable addresses.

Next note that in the instruction beginning at offset 0003, the constant 2 is moved to memory address 0. We know from examining the corresponding assembly language input file that this is the instruction that moves 2 into variable `i`. The assembler has left a place-holder of 00000000 in the object file where the linker will have to fill in the actual 32-bit address of symbol `i` once that is known. C705 is the x86 machine language opcode for the MOVL instruction where the source addressing mode is Immediate and the destination mode is Absolute. The next 4 bytes are the destination address and the final 4 bytes of the instruction are the source operand (which is in Intel-style, or "little-endian" byte order).

Here is the dump of f2.o:

```
Offset Opcodes                               Disassembly
0000   55                          pushl %ebp
0001   89E5                        movl %esp,%ebp
0003   C7050000000001000000        movl $1,0
000D   E8FCFFFFFF                  call 000E
0012   C9                          leave
0013   C3                          ret
```

Note that the placeholder in the CALL instruction is FFFFFFFC, but the disassembler decodes that as 000E. This is because the CALL instruction uses a Program Counter Relative addressing mode. The address to which execution jumps is the operand in the instruction added to the value of the Program Counter register *corresponding to the byte beyond the last byte of the CALL instruction*. In two's complement, FFFFFFFC is -4, therefore the CALL appears to be to the instruction at offset 000E. Of course, all of this is meaningless since it is just a placeholder that will be overwritten by the linker. Nonetheless, it reminds us that there are different **Relocation Types** depending on the addressing mode being used.

The symbol and relocation tables for the two object files will look something like this:

```
f1.o:
                   SYMBOL TABLE
NAME    TYPE         VALUE
f       func         0 in .text
i       variable     [reference]
                   RELOCATIONS TABLE
SYMBOL=i   OFFSET=0005  LENGTH=4  RELTYPE=ABSOLUTE



f2.o:
                   SYMBOL TABLE
NAME    TYPE         VALUE
f       function     [reference]
i       variable     8 in .bss
main    function     0 in .text
                   RELOCATIONS TABLE
SYMBOL=i   OFFSET=0005  RELTYPE=ABSOLUTE
SYMBOL=f   OFFSET=000E  RELTYPE=PCRELATIVE
```

What ld does

The first task of `ld` is to take inventory of all of the `.o` files being presented to it. It loads in the symbol tables from all of the object files to create a unified symbol table for the entire program. There is only one global namespace for all linker symbols. This might be considered a deficiency. Let's say we have a module `foo.c` that defines a function called `calculate`. If we attempt to incorporate that module into a program written by someone else, they may have also made a function called `calculate`. It is, after all, a common name.

If there is more than one defining instance of a symbol, i.e. if a symbol is *multiply-defined,* this is generally a fatal error. Consider what would happen if a programmer accidentally included two versions of function `f` above in two different `.c` files. When `f` is called somewhere in the program, which version should be called?

To ameliorate this problem of flat global linker namespace, a convention exists that one should prepend to one's global variable and function names a reasonably unique prefix. Therefore, we might call our function `foo_calculate`. This is less likely to conflict with another name. It isn't a perfect solution, but it works fairly well in reality.

In languages like C++, which takes name overloading to an extreme, the identified names used in the source code are **name-mangled** to be non-conflicting when placed into the global linker namespace.

Global variable and function names that are intended to remain private to the `.c` file in which they are declared should be protected with the `static` storage class (see below). `static` symbols still require the assistance of the linker to be relocated. However, the use of `static` causes the compiler and the assembler to flag that symbol as a LOCAL symbol. The linker will then enter the symbol into a private namespace just for the corresponding object file, and the symbol will never conflict with symbols from other object files.

There are rare cases where it is useful to deliberately redefine a symbol. For instance, we may need to change how a piece of code, available only in library or object file form, calls another function. This all falls under the heading of "wild and crazy ld tricks" and will not be discussed further. Just remember that any duplicate symbol definitions are generally wrong.

There is one antiquated exception which `ld` still honors: the so-called COMMON block, which is a hangover from FORTRAN. If there are multiple defining instances of a variable in the bss section (i.e. a global variable without an initializer), `ld` will look the other way. This is because there really isn’t a fatal conflict. As long as the definitions all give the same size and all agree that it is a bss symbol, there’s no worry about which definition is the correct one. All are equivalent. So, if one accidentally declares a global uninitialized variable twice, the program will compile fine. It is sloppy, however.

Frequently symbols have a defining instance, but they are never actually referenced. For example, the programmer may write a function, but never call it, or declare a global variable, yet never use it in an expression. The linker doesn’t care about this.

However, it cares deeply about the opposite case: a symbol that is referenced (by appearing in a relocations table) but which has no defining instance. Such a situation makes it impossible for the linker to complete its task of creating an absolute executable file with no dangling references. Therefore, it will stop with a fatal error, reporting the undefined symbol and the object file or files in which it is referenced.

Once all of the symbol definitions and references are resolved, `ld` will assemble the `a.out` file by concatenating all of the text sections of all of the object files, forming the single `.text` section of the `a.out`. In doing so, the text sections are relocated. An instruction which was at offset 10 in a particular object file’s text section may now be at offset 1034 in the `a.out`.

Furthermore, `ld` has a concept of the `load address` of the program, i.e. the memory address at which the first byte of the `.text` section of the `a.out` will be loaded. This knowledge is part of configuring `ld` for a particular operating system and processor type, and is not normally something that the programmer needs to worry about. The load address is also placed in the `a.out` file so the operating system is sure to load the program at the address for which it was linked. Knowing the size of each object file's `.text` section, `ld` can calculate the absolute address that it will occupy in the final image, and can thus complete the symbol table, assigning absolute values to each text symbol.

A similar process is undertaken for the `data` and `bss` sections. `ld` assigns an absolute address to each `data` or `bss` symbol, based on the configured starting address of the `data` and `bss` memory regions. In the case of `data` symbols, the initializers contained in the individual object files are concatenated, forming the `.data` section of the `a.out`, which will thus contain the byte-for-byte image of what that section will look like when the program starts. For `bss` symbols, there are obviously no initializers. `ld` keeps track of the total number of `bss` bytes needed, and places that information into the `a.out` header so the operating system can allocate the memory when the program is loaded. During this process, each `data` or `bss` symbol is assigned an absolute address in the symbol table.

Now `ld` concatenates the various object files, processing the relocation records, replacing each "placeholder" with the actual, absolute address that the associated symbol is now known to have. In the case of Program-Counter-Relative relocation types, the proper offset is calculated with respect to the absolute address of the placeholder in question.

Libraries

When you write a C program, you expect certain functions, such as `printf`, to be available. These are supplied in the form of a **library**. A library is basically a collection of `.o` files, organized together under a common wrapper format called a `.a` file. It is similar to a "tar" or "zip" archive (although no compression is provided).

Convention is that a library *ZZZ* is contained in the library file: `libZZZ.a`. Using the `-l` option to `cc` tells the `cc` program to ask the linker to link with the specified library. For example: `cc myprog.c -lm` asks for the system library file `libm.a` (the math library) to be additionally linked. By default, `cc` always links the standard C library `libc.a`. These libraries are located in a system directory, typically `/usr/lib`.

Although a library embodies a collection of object files, the behavior when linking to a library is slightly different than if one just linked in all of the object files individually. In the latter case, the `.text` and `.data` sections of each object file would wind up in the `a.out`, regardless of whether or not anything in a particular object file was actually used. With a library, `ld` builds a symbol table of all the object files in the library, and then only selects those object files that are actually needed for inclusion into the executable.

Dynamic Linking

When dynamic (shared) libraries are used, there are two parts to the linkage process. At compile time, `ld` links against the dynamic library (which has a `.so` extension in UNIX) for the purpose of learning which symbols are defined by it. However, none of the code or data initializers from the dynamic library are actually included in the a.out file. Instead, `ld` records which dynamic libraries were linked against.

At execution time, the second phase takes place before the `main` gets invoked. A small helper program called `ld.so` (typically, actual names vary depending on operating system build) is loaded into the process address space by the kernel and executed. The intervention of `ld.so` is made possible by the binary interpreter feature of the ELF (Extensible Linker Format) a.out format. Like the #! method for shell script (and other languages) interpreters, a flag in the a.out header informs the kernel that it needs to actually load `ld.so`, which is passed the absolute pathname of the `a.out` file as an argument. `ld.so` then does a series of mmap calls to establish the text, data and bss regions for the conventional (statically linked) part of the program, based on the a.out header information. It then locates each of the required dynamic libraries (also information from the header) and for each one, mmaps that library's text, data and bss (if needed) regions.

When using `strace` on Linux to trace system calls and we are tracing the exec of a dynamically linked program, we'll see the execve of that program. We won't see `ld.so` getting loaded because that happens as an internal kernel function, not a system call. Then next we'll see the various opens and mmaps.

Some additional magic is required to link the static and the dynamic portions of the executable. This aspect is extremely specific to the operating system and processor architecture. In general, for functions which are defined in a shared library, dummy stubs are provided in what is known as a **Procedure Linkage Table**. These stubs contain jumps to addresses which are filled in by the dynamic linker. This allows the static code to make function calls without being dynamic-aware. Likewise, a table known as the **Global Offset Table** provides transparent access to any global variables which a shared library exports. The dynamic libraries themselves need to be compiled in such as way that they are capable of being loaded at any virtual address. This is done with the `-fpic` (position independent code) flag to the compiler. The `ld.so` dynamic linker fills in the entries in the PLT and GOT based on the actual addresses that got assigned at run-time. The program is now completed linked and ready to execute, so `ld.so` jumps to the `_start` address contained in the a.out header. Since ld.so is designed to work in concert with the C library, this start function (the code of which is located in the text region associated with `libc.so`) does an munmap to destroy the text region that had contained the `ld.so` code.

There is a slight performance penalty when using dynamic libraries. In additional to the delay in executing the program imposed by needing to load and link the libraries, calls to functions in the dynamic library require an additional branch, and accesses to global symbols from within the shared library must use a position-independent addressing method, which generally requires an indirect addressing mode.

One down-side of dynamic linking is that a missing dynamic library (".so" files in UNIX terms, or "DLL"s in Windows-speak) will prevent a program from running, even if it was complete and correct at compile time. However, this disadvantage is more than outweighed by the reduction in executable size and the convenience of being able to correct errors in system libraries on the fly, without having to locate and recompile each and every executable based on the faulty library.

The program `ldd` can be used to debug dynamic library resolution problems. This example shows the dynamic libraries that are needed for `ls` on a 32-bit Linux system:

```
$ ldd /bin/ls
        linux-gate.so.1 =>  (0xb77c4000)
        librt.so.1 => /lib/librt.so.1 (0xb77b6000)
        libacl.so.1 => /lib/libacl.so.1 (0xb77ad000)
        libc.so.6 => /lib/libc.so.6 (0xb765b000)
        libpthread.so.0 => /lib/libpthread.so.0 (0xb7642000)
        /lib/ld-linux.so.2 (0xb77c5000)
        libattr.so.1 => /lib/libattr.so.1 (0xb763b000)
```

The numbers in parentheses are the addresses at which each of the dynamic libraries would load if the program were actually being executed.

Under certain circumstances, such as security-sensitive applications, it may be appropriate to use static linkage. This ensures that the program will always be able to run, and will always run with exactly the expected code.